UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Ingeniería del Software e Inteligencia Artificial

TESIS DOCTORAL

Un enfoque metalingüístico al procesamiento de documentos XML

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Bryan Temprado Battad

Director

José Luis Sierra Rodríguez

Madrid, 2016

© Bryan Temprado Battad, 2015

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE INGENIERÍA DEL SOFTWARE E INTELIGENCIA ARTIFICIAL

TESIS DOCTORAL

Un Enfoque Metalingüístico al Procesamiento de Documentos XML

Memoria para optar al grado de Doctor

Presentada por

Bryan Temprado Battad

Director

José Luis Sierra Rodríguez

Madrid, 2015

Un Enfoque Metalingüístico al Procesamiento de Documentos XML

TESIS DOCTORAL

Bryan Temprado Battad

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2015

Agradecimientos

Tras interminables horas de trabajo, y el esfuerzo de más de cuatro años, finalmente doy por finalizada esta tesis. Espero que las aportaciones que en ella residen sean de gran utilidad.

Quiero dar mi máximo agradecimiento a José Luis por su dirección y por toda la ayuda que me ha proporcionado, y también, hacer una mención especial a Alberto y Antonio por sus importantes contribuciones.

Por último, quiero agradecer a mi familia por todo su apoyo.

A todos ellos: -Gracias-

Este trabajo de tesis ha sido financiado por la beca y contrato Predoctoral UCM: BE45/10, convocatoria 9-7-2010.

Resumen

Esta tesis defiende un enfoque metalingüístico al desarrollo de aplicaciones de procesamiento XML, según el cual estas aplicaciones se conciben como clases particulares de procesadores de lenguaje, se describen utilizando formalismos de especificación de alto nivel orientados a la implementación de lenguajes informáticos, y se generan automáticamente a partir de dichas especificaciones.

La tesis comienza realizando un análisis unificado de las propuestas más relevantes al desarrollo dirigido por lenguajes de aplicaciones de procesamiento XML realizadas en el Grupo de Investigación en Ingeniería de Lenguajes Software y Aplicaciones (ILSA) de la Universidad Complutense de Madrid (UCM), tanto aquellas basadas en esquemas de traducción, como aquellas basadas en gramáticas de atributos. Como resultado de este análisis, se identifican las dos principales limitaciones de estas propuestas: (i) no abordar la relación existente entre gramáticas específicas para el procesamiento y gramáticas documentales, y (ii) no abordar adecuadamente la especificación modular de tareas complejas. Una vez identificadas estas limitaciones, la tesis se centra en paliar las mismas.

Para resolver la limitación relativa a la relación entre gramáticas específicas para el procesamiento y gramáticas documentales, la tesis propone un método para la formulación de gramáticas específicas para el procesamiento, que comienza fomentando la abstracción de la gramática documental en forma de una gramática EBNF que capture las características estructurales esenciales de la misma. Seguidamente propugna realizar gramaticalmente cada expresión regular en dicha gramática mediante una gramática BNF no autoembebible. Como resultado, es posible comprobar automáticamente si la gramática específica para el procesamiento se ha obtenido correctamente (en este caso diremos que dicha gramática es *conforme* con la gramática EBNF que abstrae a la gramática documental). Efectivamente, la comprobación de la conformidad se reduce a la comprobación de *n* equivalencias entre expresiones regulares y gramáticas no autoembebibles (una por cada no terminal *de núcleo* en la gramática EBNF). Para comprobar dicha equivalencia entre expresiones regulares y gramáticas no autoembebibles, la tesis propone un método novedoso basado en la traducción de las gramáticas no autoembebibles a expresiones regulares, así como en el uso del método de las *derivadas parciales* para traducir expresiones regulares en autómatas equivalentes. Este método de comprobación de la conformidad ofrece ventajas significativas de cara: (i) a la facilidad de diagnóstico en caso de que la comprobación de la equivalencia falle, al poder proporcionar, aparte de cadenas testigo, una justificación basada en expresiones regulares, y (ii) a una implementación con un alto grado de pereza. En esta tesis hemos desarrollado, así mismo, una generalización del método de traducción de gramáticas no autoembebibles a expresiones regulares que puede operar sobre gramáticas arbitrarias. El resultado, en este caso, no será ya una expresión regular, sino una gramática EBNF equivalente. Esto permite aplicar el criterio de conformidad directamente de gramática BNF a gramática BNF. Dicha generalización se ha plasmado en una herramienta denominada *Grammar Equivalence Checker*, que ha permitido evaluar la eficacia y usabilidad del método con resultados positivos.

La resolución de la limitación relativa a la especificación modular de tareas de procesamiento complejas se ha llevado a cabo formulando un nuevo modelo de gramática de atributos: las *gramáticas de atributos multivista*. Dichas gramáticas se organizan en *vistas gramaticales*, entendidas como fragmentos de gramáticas de atributos formuladas sobre gramáticas potencialmente diferentes a las del resto de los fragmentos, pero conformes con una gramática EBNF de base, que abstrae, a su vez, a la gramática documental. De esta forma, los árboles sintácticos asociados por las distintas vistas a un documento comparten los no terminales de *núcleo* de la gramática EBNF, y, por tanto, pueden combinarse en una única estructura de bosque de análisis sintáctico, uniéndose a través de los terminales y de dichos no terminales de núcleo. Es posible, entonces, generalizar el proceso de evaluación semántica para que, en lugar de operar sobre árboles de análisis sintáctico, opere sobre estos bosques. Dado que en cada vista es posible utilizar los atributos de los no terminales de núcleo introducidos en otras vistas, es posible interrelacionar dichas vistas no sólo sintácticamente, sino también semánticamente. En esta tesis se han definido mecanismos que permiten operacionalizar de manera efectiva el modelo, tanto en la fase de análisis, como en la fase de evaluación semántica.

En lo referente a la fase de análisis: (i) se ha desarrollado un algoritmo novedoso, denominado MVLR, que lleva a cabo el análisis sintáctico simultáneo de un documento con respecto a *n* gramáticas conformes entre sí mediante el desarrollo de *n* procesos de análisis LR en paralelo, (ii) se ha experimentado con un método heurístico basado en los algoritmos de análisis LR generalizados (GLR), y (iii) se ha desarrollado, por último, una adaptación del algoritmo MVLR, denominada MVGLR, que adapta las ideas de gestión de pilas simultáneas utilizadas en los métodos GLR para evitar redundancias durante el proceso de análisis.

En lo referente al método de evaluación semántica, se ha adaptado un método de evaluación bajo demanda convencional para su operación sobre bosques de análisis atribuidos.

Para demostrar la viabilidad del enfoque, se ha desarrollado una herramienta denominada XLOP3, que permite llevar a cabo la especificación y generación de tareas de procesamiento de documentos XML mediante gramáticas de atributos multivista. Así mismo, XLOP3 se ha utilizado para desarrollar un sistema denominado <eMU>, orientado a la generación de videojuegos educativos a partir de documentos XML. El carácter no trivial de este sistema ha arrojado evidencia positiva respecto a la utilidad práctica del enfoque.

Abstract

This Ph.D. dissertation encourages a meta-linguistic approach to the development of XML processing applications. According to this approach, each application is conceived as a kind of language processor. Thus, these applications can be described using high-level specification formalisms for computer languages. In addition, these applications can be automatically generated from these specifications.

The dissertation begins with a unified analysis of the most relevant proposals on language-driven development of XML processing applications carried out in the ILSA (Implementation of Language-driven Software and Applications) research group at UCM (Complutense University of Madrid, Spain): translation schemata-based proposals, and attribute grammars-based ones. As a result of this analysis, this dissertation identifies the two main shortcomings of the ILSA language-driven development approaches to XML processing: (i) to do not address the existing relationships between processing-specific grammars and document grammars, and (ii) to do not address in a suitable way the modular specification of complex processing tasks. Once these limitations are identified, the dissertation focuses on alleviating those.

To solve the limitation concerning the relationships between processing-specific grammars and document grammars, this dissertation proposes a method that promotes the abstraction of the document grammar in an EBNF grammar that models the essential structural features. Then each regular expression in the EBNF grammar is implemented using a non-self-embedding context-free grammar. As a consequence, it is possible to check automatically whether the processing-specific grammar has been formulated in a sound way (it this case this grammar will be said to *conform* to the document grammar-abstracting EBNF grammar). Indeed, it is possible to reduce this conformance checking method to *n* problems of checking the equivalency of a regular expression and a non-self-embedding grammar (one for each *core* non-terminal in the EBNF grammar). In order to check whether this equivalency between regular expressions and non-self-embedding grammars holds, the dissertation proposes a method based on the translation of non-self-embedding grammars into equivalent regular expressions using a new algorithm developed for this purpose, and on the translation of regular expressions into non-deterministic finite automata using the *partial derivatives* method. The main advantages of this method are: (i) to enhance the diagnose capabilities when the equivalency checking fails, since it is able to provide, in addition to a witness string, justifications based on regular expressions, and (ii) to enable a highly lazy implementation. In addition, in this dissertation we have generalized the method to work with arbitrary, no necessarily non self-embedding, grammars. When applied to a self-embedding grammar, the result is no longer a single regular expression but an equivalent EBNF grammar. It enables the application of the conformance checking criterion for BNF to BNF grammars. This generalization is the basis of a tool called *Grammar Equivalency Checker*, developed in this dissertation, and which has enabled the assessment of the conformance checking method with very promising results.

As for the limitation concerning the modular specification of complex processing tasks, it has been addressed by formulating a new attribute grammar model: *multiview attribute grammars*. Such grammars are organized in sets of *grammar views*, which are conceived as attribute grammar fragments formulated on context-free grammars potentially distinct to those of the other fragments, but still conforming to the base EBNF grammar that abstracts the document grammar. In this way, the parse trees for a document associated with the different views share the core non-terminals defined in the EBNF grammar, and, therefore, those can be combined in a single parse forest structure by gluing those together using the leaves (terminal symbols) and the inner nodes for core non terminals. As a consequence, it is meaningful to generalize the semantic evaluation process for operating on parse forests instead of on parse trees. Since in each view it is possible to use the core non-terminals attributes that were introduced in other views, it is possible to relate the views not only from the point of view of the syntax, but also for that of the semantics. This dissertation introduces mechanisms to operationalize the model in an effective way, both in the analysis and in the semantic evaluation stages.

Concerning the analysis stage: (i) a new algorithm, called MVLR, has been developed that enables to simultaneously parsing a document with respect to *n* conformant grammars, maintaining *n* LR parsing processes in parallel, (ii) an experimentation with a heuristic method based on generalized LR (GLR) methods has been carried out, and (iii) an adaptation of the MVLR algorithm called MVGLR has been developed that adapts the management techniques of simultaneous stacks used in the GLR methods in order to avoid redundancy during parsing.

As for the semantic evaluation method, this dissertation proposes to adapt a conventional demand-driven evaluation method to work on parse forests.

Finally, in order to demonstrate the feasibility of the approach, a tool called XLOP3 has been developed, which enables the specification of XML processing tasks with multiview attribute grammars, as well as the automatic generation of the applications from these high-level specifications. In addition, XLOP3 has been used for developing a system called <eMU>, which makes it possible to generate educational games from XML documents. The non-trivial nature of this system provides positive evidence to the practical applicability of the approach.

Índice

Capítulo 1
Introducción y Objetivos

1.1 Introducción

XML (eXtensible Markup Language) [Bray et al. 2008], un metalenguaje basado en el estándar SGML (Standard Generalized Markup Language) [Goldfarb 1991] y propuesto a finales del siglo pasado por el Word Wide Web Consortium (W3C), se ha convertido actualmente en un pilar fundamental de cualquier sistema de información. Efectivamente, XML permite estructurar la información intercambiada entre los distintos componentes de un sistema informático en términos de *documentos* que pueden ser leídos por personas y procesados por programas. Para ello, XML permite codificar tales documentos mediante etiquetas o marcas que explicitan la estructura lógica de los mismos. XML permite, así mismo, definir y utilizar el vocabulario de etiquetas (el *lenguaje de marcado*) más apropiado para cada tipo de documentos. De esta forma, la aplicación de XML a un dominio problema particular es fundamentalmente *metalingüística*, ya que consiste en gran medida en diseñar un lenguaje de marcado apropiado para estructurar la información que surge en dicho dominio. Dicho diseño se plasma habitualmente en una *gramática documental* que describe la sintaxis del lenguaje, utilizando un formalismo gramatical adecuado (p.e., el formalismo de las DTDs, *Document Type Definitions*, integrado en el propio XML).

Una característica básica del marcado XML es que éste es *descriptivo*: permite describir estructuras, pero no dicta la forma en la que dichas estructuras deben ser procesadas. Por tanto, el marcado descriptivo propugnado por XML mantiene la separación explícita de la descripción de la estructura de la información y los posibles mecanismos de procesamiento de la misma [Coombs et al. 1987]. XML en sí no define mecanismos para llevar a cabo dichos aspectos de procesamiento. Para tal fin se han propuesto, sin embargo, una serie de tecnologías y enfoques complementarios que permiten añadir procesamiento a los documentos XML.

A este respecto, el enfoque metalingüístico intrínseco a la aplicación de XML para modelar tipos de documentos electrónicos no tiene, sin embargo, una correspondencia directa a nivel del procesamiento de los documentos resultantes. Efectivamente, las tecnologías de procesamiento de documentos XML más usuales entienden los documentos XML como *datos estructurados* (árboles, secuencias de eventos, cadenas de elementos de información) en lugar de como cadenas de lenguajes formales apropiados. Como consecuencia, dichas tecnologías no aprovechan la naturaleza lingüística intrínseca a los lenguajes de marcado.

A este respecto, una de las líneas de investigación más activas llevadas a cabo en el grupo ILSA (Grupo de Investigación en Ingeniería de Lenguajes Software y Aplicaciones) de la UCM, grupo en cuyo contexto se ha llevado a cabo este trabajo de tesis, consiste en el desarrollo de

enfoques *dirigidos por lenguajes* al procesamiento de documentos XML. Estos enfoques, al contrario que las tecnologías de procesamiento de documentos XML más convencionales, enfatizan el carácter lingüístico de los vocabularios de marcado XML para dominios específicos. Desde este punto de vista, tales vocabularios se entienden como lenguajes formales, y el procesamiento de los documentos resultantes se entiende como el procesamiento de las frases pertenecientes a lenguajes formales apropiados. Por tanto, y desde este punto de vista, las aplicaciones de procesamiento XML pueden entenderse como un tipo particular de *procesador de lenguaje*. Este hecho justifica, entonces, el empleo de las diferentes tecnologías utilizadas en el desarrollo de procesadores de lenguaje para llevar a cabo la construcción, mantenimiento y evolución de este tipo de aplicaciones.

De esta forma, la presente tesis se adhiere a esta línea de investigación. En concreto, esta tesis complementa y continúa los trabajos sobre procesamiento de documentos XML ya abordados en una tesis anterior realizada en el grupo, la del Prof. Antonio Sarasa "Desarrollo de Aplicaciones XML mediante Herramientas de Construcción de Procesadores de Lenguaje" [Sarasa 2012]. Para ello, la presente tesis realiza un análisis crítico de los enfoques dirigidos por lenguajes desarrollados en los últimos 8 años en el grupo ILSA, enfatizando también las contribuciones a los mismos del propio autor de la misma. Como resultado de dicho análisis, contrasta y resuelve algunas de las principales limitaciones no abordadas por los trabajos previos.

A continuación, se describen con más detalle los objetivos planteados para esta tesis doctoral (sección 1.2). Seguidamente se describe la estructura de esta memoria de tesis (sección 1.3). Por último, se describen y contextualizan también las principales publicaciones asociadas a la misma (sección 1.4).

1.2 Objetivos de la tesis

El objetivo global de esta tesis es contribuir a asentar el enfoque dirigido por lenguajes al desarrollo de aplicaciones de procesamiento XML, mostrando la viabilidad y conveniencia de entender el mismo como una actividad eminentemente *metalingüística* (es decir, como una actividad de formulación de lenguajes, y de desarrollo de procesadores para dichos lenguajes mediante formalismos específicos para llevar a cabo dicho desarrollo). Este objetivo general se estructura, de esta forma, en los tres siguientes objetivos concretos:

- *Caracterización del desarrollo de aplicaciones de procesamiento XML como una actividad metalingüística*. La concepción metalingüística del desarrollo de aplicaciones de documentos XML presupone la posibilidad de describir dichas aplicaciones utilizando formalismos habitualmente usados en el desarrollo de procesadores de lenguaje (p.e., esquemas de traducción, gramáticas de atributos). De esta forma, como primer objetivo de esta tesis se plantea la caracterización precisa de cómo llevar a cabo dicha descripción. Para ello, se analizarán los trabajos realizados en este sentido en el grupo ILSA, incidiendo en diferentes aspectos a cuya formulación y consolidación, el autor de esta tesis ha contribuido desde su incorporación al grupo en 2007: modelos de proceso

de desarrollo y especificación declarativa de programas de procesamiento de documentos XML mediante formalismos específicos para el desarrollo de procesadores de lenguaje.

- *Caracterización de la conformidad entre gramáticas documentales y gramáticas específicas del procesamiento y formulación de un método efectivo para comprobar dicha conformidad*. Uno de los aspectos clave en el desarrollo dirigido por lenguajes de aplicaciones de procesamiento XML es el uso de *gramáticas incontextuales* específicamente diseñadas para orquestar el *procesamiento dirigido por la sintaxis* de los documentos. Dichas gramáticas estarán orientadas al procesamiento, por lo que diferirán de las gramáticas documentales utilizadas para definir los correspondientes lenguajes de marcado XML. No obstante, es necesario establecer mecanismos que permitan comprobar que, en un sentido apropiado, tales gramáticas específicas para el procesamiento *realizan* a las gramáticas documentales que definen los lenguajes de marcado. En este caso diremos que las gramáticas específicas para el procesamiento son *conformes* con las correspondientes gramáticas documentales. Es por ello que, en esta tesis, planteamos como segundo objetivo el caracterizar criterios de conformidad adecuados entre ambos tipos de gramáticas (gramáticas específicas para el procesamiento y gramáticas documentales), así como formular métodos efectivos para comprobar automáticamente dichos criterios.

- *Mejora de los mecanismos de modularidad en la descripción metalingüística de aplicaciones de procesamiento de documentos XML*. El desarrollo de aplicaciones de procesamiento de documentos XML complejas requiere, así mismo, incorporar mecanismos de modularización adecuados en los instrumentos utilizados en dicho desarrollo. Esto también es cierto en el caso de que en dicho desarrollo se utilicen formalismos declarativos de alto nivel, como aquellos basados en las notaciones habituales utilizadas durante el desarrollo de un procesador de lenguaje. A este respecto, en el grupo ILSA se han propuesto ya algunos de estos mecanismos que, sin embargo, presentan una limitación básica: las especificaciones deben compartir la misma gramática específica para el procesamiento. Dado que las gramáticas específicas para el procesamiento se adecúan a procesamientos concretos, la adopción de una gramática común puede constreñir la descomposición de tareas de procesamiento complejas en subtareas más simples, ya que cada subtarea puede requerir para su correcta realización una sintaxis específica, que puede diferir, a su vez, de la requerida por las otras tareas. Es por ello que en esta tesis planteamos como tercer objetivo el introducir mecanismos de modularización que admitan la coexistencia, en una misma especificación, de múltiples sintaxis conformes entre sí.

1.3 Estructura de la memoria

Esta memoria de tesis, aparte de este introductorio, integra los siguientes cinco capítulos:

- El Capítulo 2 contiene un estudio de los aspectos relativos al estado del arte más relevantes para el desarrollo de esta tesis. Dicho capítulo comienza realizando una revisión de los aspectos más relevantes para esta tesis sobre la construcción de procesadores para lenguajes informáticos, y de los formalismos declarativos utilizados para su especificación y generación que resultan más relevantes de cara a la misma (en particular, esquemas de traducción y gramáticas de atributos). El capítulo introduce, así mismo, el metalenguaje XML, los formalismos de gramáticas documentales utilizados para definir lenguajes de marcado basados en XML, y las tecnologías convencionales habitualmente utilizadas para el procesamiento de documentos XML (tanto de carácter específico, como de propósito general). Finalmente, el capítulo revisa distintos enfoques lingüísticos al procesamiento de documentos XML, enfoques que utilizan y adaptan al dominio XML los formalismos para la especificación de procesadores de lenguaje referidos anteriormente: esquemas de traducción, y gramáticas de atributos.

- El Capítulo 3 describe los distintos enfoques al desarrollo dirigido por lenguajes de aplicaciones de procesamiento de documentos XML llevados a cabo en el seno del grupo ILSA, y en los cuáles el autor de esta tesis ha participado activamente. Para ello, se lleva a cabo un análisis unificador que enfatiza los aspectos de modelo de proceso de desarrollo y de especificación declarativa de aplicaciones promovidos por dichos enfoques. En concreto, se describen enfoques orientados a la utilización de esquemas de traducción, así como al soporte de los mismos mediante herramientas convencionales de construcción de procesadores de lenguaje (p.e., YACC, JavaCC, CUP, ANTLR). Así mismo, se describen también enfoques basados en gramáticas de atributos, introduciendo la herramienta XLOP, una meta-meta-herramienta que toma como entrada una gramática de atributos que especifica una tarea de procesamiento de documentos XML, y produce como salida una implementación CUP de dicha especificación (CUP es un generador de traductores ascendentes en Java, tipo YACC). Por tanto, el desarrollo de este capítulo está directamente relacionado con el primero de los objetivos planteado en la sección anterior.

- El Capítulo 4 aborda el problema de la conformidad entre gramáticas específicas para el procesamiento XML y las gramáticas documentales utilizadas en la definición de los vocabularios XML de marcado descriptivo. Para ello, propone abstraer las características estructurales básicas de una gramática documental en términos de una gramática EBNF *de base*. Así mismo, propone formas de caracterizar gramaticalmente, mediante subgramáticas BNF, las expresiones regulares que definen los no terminales de dicha gramática EBNF de base, de tal forma que se garantice que el problema de comprobar la equivalencia entre dichas subgramáticas y las correspondientes expresiones regulares sea efectivamente resoluble. Como consecuencia, se establece el criterio de comprobación de la conformidad como la comprobación de la equivalencia

entre dichas subgramáticas y las correspondientes expresiones regulares. Se estudia, así mismo, un mecanismo para llevar a cabo dicha comprobación de la equivalencia que tenga en cuenta el factor humano en el proceso de comprobación de la conformidad, siendo capaz de ofrecer diagnósticos significativos en caso de que dicho proceso devuelva resultados negativos. Finalmente, se evidencia cómo dicho mecanismo puede generalizarse para transformar gramáticas BNF arbitrarias en gramáticas EBNF equivalentes, lo que permite generalizar, a su vez, el criterio de comprobación de la conformidad entre gramáticas para tener en cuenta gramáticas BNF arbitrarias. Se muestra, así mismo, algunos resultados de evaluación empírica de la utilidad del método con desarrolladores. Este capítulo aborda, por tanto, el segundo de los objetivos planteados.

- El Capítulo 5 aborda el problema de la mejora de la modularidad. Para ello, se centra en la especificación declarativa de tareas de procesamiento de documentos XML mediante gramáticas de atributos, y propone una extensión del formalismo básico de las gramáticas de atributos que se ha denominado *gramáticas de atributos multivista*. Una gramática multivista permite descomponer una gramática de atributos en fragmentos, fragmentos que se formulan sobre gramáticas incontextuales que pueden diferir entre sí, pero que deben ser conformes con una gramática EBNF de base. De esta forma, el formalismo resuelve la limitación asociada a una gramática específica para el procesamiento común a la que se ha hecho alusión anteriormente. Se desarrolla, así mismo, un modelo de ejecución para gramáticas de atributos multivista apropiado. Para ello se formulan distintos algoritmos de análisis sintáctico simultáneo con respecto a *n* gramáticas incontextuales no ambiguas conformes entre sí. Se formula, así mismo, una estrategia de evaluación semántica bajo demanda que opera sobre los *bosques sintácticos* (las estructuras que resultan de amalgamar en una única los árboles de análisis sintáctico asociados con cada gramática). Se describe también XLOP3, una evolución del sistema XLOP que implementa el formalismo de las gramáticas de atributos multivista, y que se ha desarrollado como parte de esta tesis para justificar la viabilidad del formalismo como mecanismo para orquestar el enfoque metalingüístico al desarrollo de aplicaciones de procesamiento de documentos XML. Se describe, por último, el desarrollo con XLOP3 de <eMU>, un sistema para la generación de juegos educativos a partir de documentos XML que permite justificar, sobre un caso de estudio no trivial, la utilidad práctica de la propuesta basada en gramáticas de atributos multivista. Este capítulo aborda, por tanto, el tercero de los objetivos planteados.

- El Capítulo 6, por último, presenta las conclusiones obtenidas como consecuencia del desarrollo de la tesis, y propone posibles líneas de trabajo futuro para continuar las líneas de investigación abiertas en la misma.

1.4 Publicaciones asociadas

A continuación, se enumeran las publicaciones asociadas a esta tesis. Dado que, tal y como se ha indicado anteriormente, esta tesis es una continuación de la tesis del Prof. Antonio Sarasa "Desarrollo de Aplicaciones XML mediante Herramientas de Construcción de Procesadores de Lenguaje" [Sarasa 2012], varias de estas publicaciones contienen aportaciones de ambos trabajos. Es por ello que las publicaciones se dividirán en tres grupos:

- Publicaciones que contienen mayoritariamente aportaciones propias de la presente tesis "Un Enfoque Metalingüístico al Procesamiento de Documentos XML".
- Publicaciones que contienen aportaciones significativas de ambas tesis.
- Publicaciones que contienen mayoritariamente aportaciones propias de la tesis del Prof. Sarasa "Desarrollo de Aplicaciones XML mediante Herramientas de Construcción de Procesadores de Lenguaje".

En relación con el primer grupo, publicaciones que contienen mayoritariamente aportaciones propias de la presente tesis, caben destacar las tres siguientes:

- Bryan Temprado-Battad, Antonio Sarasa-Cabezuelo, José-Luis Sierra: Managing the Production and Evolution of e-learning Tools with Attribute Grammars. Proc. of the 10th IEEE International Conference on Advanced Learning Technologies: 427-431. IEEE Computer Society. 2010 (**CORE B**).
- Bryan Temprado-Battad, Antonio Sarasa-Cabezuelo, José-Luis Sierra: Modular Specifications of XML Processing Tasks with Attribute Grammars Defined on Multiple Syntactic Views. Fifth International Workshop on Flexible Database and Information Systems Technology - FlexDBIST 2010. Proc. of the Database and Expert Systems Applications, DEXA, International Workshops: 337-341. 2010. IEEE Computer Society (**CORE B**).
- Bryan Temprado-Battad, Antonio Sarasa-Cabezuelo, José-Luis Sierra: Checking the Conformance of Grammar Refinements with Respect to Initial Context-Free Grammars. Proc. of the Federated Conference on Computer Science and Information Systems - FedCSIS 2011: 887-890. IEEE Computer Society. 2011.

En relación con el segundo grupo, publicaciones que contienen aportaciones significativas de ambos trabajos, caben destacar las dos siguientes:

- Antonio Sarasa-Cabezuelo, Bryan Temprado-Battad, Daniel Rodríguez-Cerezo, José-Luis Sierra: Building XML-driven application generators with compiler construction tools. Computer Science and Information Systems, 9(2): 485-504. 2012 (**IF JCR 2012: 0.549**).
- Antonio Sarasa-Cabezuelo, Bryan Temprado-Battad, José-Luis Sierra, Alfredo Fernández-Valmayor: XML Language-Oriented Processing with XLOP. Fifth International Symposium on Web and Mobile Information Services (WAMIS 2009). 23rd International

Conference on Advanced Information Networking and Applications, AINA 2009, Workshops Proceedings: 322-327. IEEE Computer Society (**CORE B**).

Por último, en relación con el tercer grupo, publicaciones que, aun conteniendo algunos aspectos relacionados con la presente tesis, contienen mayoritariamente aportaciones propias de la tesis del Prof. Sarasa, caben destacar las dos siguientes:

- Antonio Sarasa-Cabezuelo, Bryan Temprado-Battad, José-Luis Sierra: Engineering web services with attribute grammars: a case study. ACM SIGSOFT Software Engineering Notes 36(1): 9pp. 2011.

- Antonio Sarasa-Cabezuelo, Bryan Temprado-Battad, Alberto Martinez-Aviles, José-Luis Sierra, Alfredo Fernández-Valmayor: Building an Enhanced Syntax-Directed Processing Environment for XML Documents by Combining StAX and CUP. Four International Workshop on Flexible Database and Information Systems Technology - FlexDBIST 2009. Proc. of the Database and Expert Systems Applications, DEXA, International Workshops: 427-431. 2009. IEEE Computer Society (**CORE B**).

Capítulo 2
Estado del Dominio

2.1 Introducción

A lo largo de este capítulo se presentan los diferentes temas, contenidos, procesos, conceptos y herramientas que han servido de motivación y de base al desarrollo de esta propuesta de tesis. Dado que esta tesis propone la aplicación de un enfoque dirigido por lenguajes al procesamiento de documentos XML, según el cual las aplicaciones que procesan documentos XML se entienden como un tipo específico de procesadores de lenguaje, el capítulo comienza con la exposición de conceptos básicos sobre procesamiento de lenguajes en la sección 2.2, presentando así mismo las herramientas y notaciones existentes más utilizadas para llevar a cabo la especificación de dicho procesamiento (herramientas y notaciones basadas en esquemas de traducción y gramáticas de atributos). Al estar la propuesta de tesis basada en el procesamiento de documentos XML, la sección 2.3 describe el formato de dichos documentos, los mecanismos para especificar dicho formato, y las herramientas convencionales utilizadas para su procesamiento. La sección 2.4 presenta diferentes enfoques dirigidos por lenguajes al procesamiento de documentos XML, campo en el que, como ya se ha comentado, se encuadra la presente tesis. Por último, la sección 2.5 concluye el capítulo, contextualizando la tesis en el estado del dominio presentado.

2.2 Procesadores de lenguaje y su especificación

En la actualidad, las aplicaciones informáticas requieren manejar cantidades elevadas y estructuradas de información. Esta tesis propone abordar la construcción de este tipo de aplicaciones utilizando métodos, técnicas y herramientas orientadas al procesamiento de lenguajes formales. De esta forma, los *procesadores de lenguaje*, descritos en la sección 2.2.1, pueden proporcionar una solución a la forma de interpretar y procesar esta información, que bien puede venir dada en formato documental o ser simplemente un flujo de datos. Lo que caracteriza a este tipo de procesadores, así aplicados, es que se construyen para procesar una información que atiende a un lenguaje específico estructurado alrededor de sus posibles construcciones sintácticas. Por tanto, en la sección 2.2.2 se presenta el formalismo descriptivo más utilizado en la práctica para caracterizar dichas estructuras sintácticas: las *gramáticas incontextuales*. Esta concepción implica a su vez, proponer implementaciones eficientes y específicas de *analizadores*, como se detalla en la sección 2.2.3, que soporten el reconocimiento de los distintos tipos de estructuras sintácticas. Así mismo, para poder definir el procesamiento que se desea realizar, es necesario contar con técnicas descriptivas que nos permitan establecer cómo se realizará el cómputo que caracteriza el procesamiento que se ha definido. Los *esquemas de traducción* son un ejemplo de este tipo de técnicas, que se

presentan en la sección 2.2.4 junto con las herramientas de generación de procesadores asociadas. Finalmente, en la sección 2.2.5 se presentan las *gramáticas de atributos*, otro ejemplo significativo de técnicas para la descripción de procesadores de lenguaje, que constituyen un formalismo declarativo de más alto nivel que el de los esquemas de traducción.

2.2.1 Procesadores de lenguaje

Los procesadores de lenguaje son programas que permiten la lectura, transformación y operacionalización de sentencias pertenecientes a un lenguaje específico. Los lenguajes que procesan estos programas, formalmente se definen en [Chomsky 1957] como un conjunto finito o infinito de sentencias de longitud finita construidos a partir de un conjunto finito de elementos. En el contexto de esta tesis, dichos lenguajes serán lenguajes informáticos, y, más concretamente, lenguajes de marcado XML para la descripción de documentos electrónicos.

Los ejemplos más directos de procesadores de lenguajes son los compiladores de los lenguajes de programación, como C# o Java. Estos programas realizan la transformación de un código escrito en un lenguaje de alto nivel (es decir, legible y sencillamente interpretable por los humanos), en uno equivalente a bajo nivel, comúnmente instrucciones o códigos que son interpretados por una máquina específica.

Las siguientes secciones presentan los aspectos más relevantes para esta tesis sobre procesadores de lenguaje, su especificación y su construcción.

2.2.2 Gramáticas incontextuales

La sintaxis y estructura de un lenguaje informático puede describirse formalmente mediante una *gramática incontextual* [Aho et al. 2006, Grune & Jacobs 2008], un formalismo propuesto por el lingüista norteamericano Noam Chomsky [Chomsky 1956] para la especificación de la estructura sintáctica básica de un lenguaje. Los elementos descriptivos de los que se compone una gramática incontextual son los siguientes:

- Terminales. Son símbolos sintácticos del lenguaje referidos a los *tokens*: palabras o secuencias de caracteres del lenguaje que se identifican bajo una misma clase léxica. El terminal en sí es el nombre de la clase léxica. Como ejemplo puede citarse un terminal *número*, que identifica bajo dicho símbolo *número* cualquier cadena de números, como podrían ser “10”, “20”, etc. De esta forma, un token es una correspondencia *terminal-valor* como, por ejemplo, *número*-“20”. Como notación para describir tokens emplearemos caracteres entre comillas simples o palabras en minúsculas y, entre comillas dobles, un valor instancial.

- No terminales. Denotan conjuntos de cadenas formadas por terminales. Como ejemplo, un no terminal T podría incluir la cadena de terminales *número* ‘+’ *número*, o la cadena

‘(‘ *número* ‘)’. Como notación para los no terminales emplearemos palabras con letra inicial mayúscula.

- Producciones. Se componen de cabeza y de cuerpo. La cabeza de una producción o parte izquierda está formada por un único no terminal. El cuerpo de la producción o parte derecha, describe el conjunto de cadenas que forman la cabeza mediante una secuencia finita de terminales y/o de no terminales. Mediante producciones se consigue describir la estructura sintáctica del lenguaje. Como notación, emplearemos el símbolo “::=” para separar los elementos cabeza y cuerpo entre sí. Por ejemplo, la producción T ::= *número* ‘+’ *número* permite reconocer bajo el no terminal T la *sentencia* o cadena de caracteres: “10+20”.

- Axioma. Es el símbolo no terminal comienzo de la descripción del lenguaje.

En la Figura 2.2.1 se muestra una gramática incontextual de ejemplo que define el lenguaje de las expresiones aritméticas con suma (mediante λ se denota la cadena vacía). Algunos ejemplos de cadenas de este lenguaje son “10+20”y “x+20”, donde en este caso, 10 y 20 son números, y “x” es un nombre de variable (tales cadenas se reconocen como tokens a nivel léxico).

1. Expresión ::= Operando RestoExp
2. RestoExp ::= ‘+’ Operando RestoExp
3. RestoExp ::= λ
4. Operando ::= variable
5. Operando ::= número

Figura 2.2.1. Gramática de ejemplo: expresiones aritméticas con suma.

Partiendo del axioma de la gramática es posible obtener las *formas sentenciales* y las *sentencias* que se derivan a partir de dicho no terminal. Las formas sentenciales son secuencias de terminales y no terminales derivables desde el axioma de la gramática, mientras que las sentencias son secuencias de terminales pertenecientes al lenguaje de la gramática. Este proceso, llamado de *derivación*, consiste en realizar un simple método de reescritura a base de sustituir símbolos no terminales (partiendo del axioma) por el cuerpo de alguna de las producciones que los definen, en sucesivas reiteraciones. Con ello se obtienen las posibles sentencias que pertenecen al lenguaje como las cadenas de terminales derivables desde el axioma. Como notación de derivación utilizaremos $S \Rightarrow^* \alpha$, donde α es una forma sentencial derivada desde el no terminal S, y la aparición de * indicará que se ha realizado el proceso de reescritura en cero o más pasos para obtener tal forma sentencial α. De manera gráfica, realizar esta operación produce un *árbol de derivación* o de *análisis sintáctico*. En la Figura 2.2.2 se muestra una sentencia que se deriva a partir de la gramática con axioma *Expresión* de la Figura 2.2.1, y el árbol de derivación o árbol de análisis sintáctico asociado.

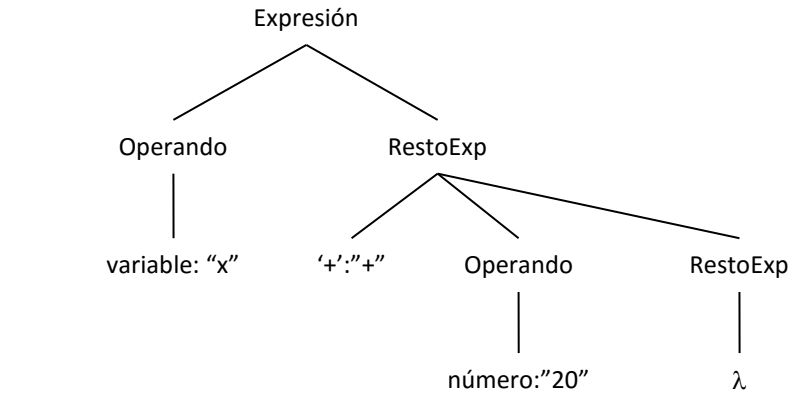


Figura 2.2.2. Árbol de derivación o de análisis sintáctico para la sentencia “x+20”.

Dentro de las gramáticas incontextuales se distinguen diferentes subclases que permiten implementar sistemáticamente distintos tipos de analizadores sintácticos. Algunos ejemplos son las clases de gramáticas LL(k), LR(k), LALR(k), etc., donde *k* son los k símbolos de *predicción* o *preanálisis* empleados por los analizadores resultantes. En la siguiente sección se revisan los tipos de analizadores sintácticos más relevantes de cara a esta tesis.

Para finalizar, es conveniente indicar que, como notaciones formales para describir gramáticas incontextuales, se utiliza comúnmente:

- La notación BNF (Backus-Naur Form) [Naur 1960]. En esta notación, los símbolos no terminales se representan como cualquier cadena de caracteres contenida entre “<” “>”, y los símbolos terminales contenidos entre comillas dobles. El cuerpo de una regla de producción viene separado de la cabeza por “::=”, y mediante el símbolo “|” se declaran varios cuerpos de producción de manera compacta para el mismo no terminal cabeza de la producción.

- La notación EBNF (Extended Backus-Naur Form), que permite emplear *expresiones regulares* más generales en las partes derechas de las producciones, en las que pueden utilizarse operaciones como la agrupación con “(“ y “)”, el cierre de Kleene “*” indicando cero o más repeticiones, “+” indicando una o más repeticiones, y “?” indicando opcionalidad.

Como ya se ha indicado anteriormente, en esta tesis los no terminales se denotarán mediante símbolos que comienzan por mayúscula, mientras que los terminales se denotarán mediante símbolos que comienzan por minúscula, lo que permitirá simplificar la notación (al omitir, por ejemplo, “<” y “>). También se emplearán comillas simples para denotar terminales que involucran símbolos especiales. Por último, mediante λ se describirá la cadena vacía, y se utilizará “|” como separador de cuerpos de producción asociados a una misma cabeza de producción. Un ejemplo de producciones siguiendo este convenio es A ::= aB | b | λ.

2.2.3 Analizadores sintácticos

Los analizadores sintácticos son programas que permiten reconocer las sentencias de un lenguaje determinado, construidos mediante la especificación aportada por una gramática incontextual. Durante el análisis que llevan a cabo, realizan la construcción implícita de un árbol de análisis sintáctico derivado de una sentencia de entrada, normalmente de dos formas diferentes: de forma ascendente o de forma descendente [Aho et al. 2006]. A continuación, se revisan ambos modelos, haciendo énfasis en el de análisis ascendente, ya que juega un papel importante en esta tesis.

2.2.3.1 Analizadores descendentes

Los algoritmos de *análisis descendente* construyen el árbol de análisis sintáctico desde la raíz hasta las hojas. La construcción comienza en el axioma de la gramática y continúa con la expansión del mismo por el cuerpo de alguna de sus producciones. A su vez, los distintos nodos pertenecientes a símbolos no terminales son expandidos por los correspondientes cuerpos de producción, y así sucesivamente hasta permanecer los símbolos terminales (o bien el símbolo especial λ, que denota la cadena vacía) como nodos hoja del árbol. Los símbolos terminales, a su vez, se corresponderán con las cadenas o *tokens* que forman la sentencia de entrada.

En el método de análisis descendente surge, de esta forma, el problema de decisión sobre qué regla de producción tomar y expandir en cada momento, dado un nodo perteneciente a un símbolo no terminal en el árbol. Como solución de cara a una implementación determinista, los analizadores descendentes realizan la construcción predictiva del árbol observando los *k* símbolos siguientes o de preanálisis de la cadena de entrada que se analiza. Por este motivo, las gramáticas que estos analizadores soportan se denominan gramáticas LL(k), en función de los *k* símbolos de preanálisis aplicados.

2.2.3.2 Analizadores ascendentes

En los algoritmos de *análisis ascendente*, al contrario que en los descendentes, la construcción del árbol de análisis sintáctico se realiza partiendo de las hojas, creando nodos padres para el símbolo no terminal cuyo cuerpo de producción se corresponda con secuencias de nodos hijos ya creados, y finalizando en el axioma o raíz del árbol. El proceso que se lleva a cabo formalmente es, dada una sentencia de entrada, reconstruir, de manera inversa (es decir, comenzando por el final y progresando en sentido inverso, hasta alcanzar el axioma) una derivación *más a la derecha* de dicha sentencia (es decir, una derivación en la que siempre se reescribe el no terminal más a la derecha de cada forma sentencial). Para ello, se utiliza una *pila de análisis sintáctico* que albergará, para cada forma sentencial αw en la derivación más a la derecha, la cadena α, que se denomina *prefijo viable*. La cadena *w* representará, por tanto, el fragmento de entrada restante por analizar. El modelo de análisis ascendente más habitual

realiza las siguientes operaciones para reconstruir en sentido inverso la derivación más a la derecha:

- *Desplazamiento*: apilar en la pila de análisis sintáctico el primer símbolo del resto de la sentencia de entrada.
- *Reducción*: desapilar de la pila de análisis sintáctico el cuerpo de una producción y apilar, en su lugar, el símbolo cabeza de dicha producción.

Estas dos posibles operaciones caracterizan a estos analizadores sintácticos como analizadores de *desplazamiento-reducción*. Cuando se produce una reducción por el axioma de la gramática y cada símbolo de la sentencia de entrada ha sido consumido, el reconocimiento de dicha sentencia habrá concluido correctamente, y se decidirá su pertenencia al lenguaje descrito por la especificación gramatical. La instrucción resultante es *aceptar*. En caso contrario se producirá un error. También es posible que se produzcan conflictos cuando el algoritmo detecta que es capaz de realizar más de una operación de las descritas anteriormente, sobre una determinada configuración de la pila. En la Figura 2.2.3 se muestra el proceso que realizaría un analizador ascendente, para la gramática de ejemplo de las expresiones aritméticas de la Figura 2.2.1, al analizar la cadena de entrada “x+20”.

Pila de análisis sintáctico	Resto de la cadena de entrada	Acción Desplazamiento-Reducción
	“x+20”	**Desplazar** variable
variable:”x”	“+20”	**Reducir** por Operando ::= variable
Operando	“+20”	**Desplazar** ‘+’
Operando ‘+’:”+”	“20”	**Desplazar** número
Operando ‘+’ número:”20”	“”	**Reducir** por Operando ::= número
Operando ‘+’ Operando	“”	**Reducir** por RestoExp ::= λ
Operando ‘+’ Operando RestoExp	“”	**Reducir** por RestoExp ::= ‘+’ Operando RestoExp
Operando RestoExp	“”	**Reducir** por Expresión ::= Operando RestoExp
Expresión	“”	**Aceptar**

Figura 2.2.3. Proceso de análisis ascendente de desplazamiento-reducción.

Las operaciones de desplazamiento y reducción realizadas por los analizadores sintácticos anteriormente descritos pueden guiarse por una *tabla de análisis sintáctico.* Para la obtención de este tipo de tablas se realiza habitualmente la construcción de un autómata finito determinista que reconozca los prefijos viables de la gramática incontextual. Existen diversos métodos para tal fin, que producen autómatas con más o menos estados y caracterizan la clase de gramáticas para las que pueden generar tablas sin conflictos. Tales métodos se denominan conjuntamente *métodos* LR. Los métodos LR más habituales son el LR(0) (funciona

para las denominadas gramáticas LR(0)), el LR(1) (funciona para las denominadas gramáticas LR(1), e implica, además, el uso de un símbolo de preanálisis durante el reconocimiento), y el LALR(1) (Lookahead LR), el más destacado y utilizado en la práctica [DeRemer 1969], que funciona para las denominadas gramáticas LALR(1) (son una clase de gramáticas más restringida que las LR(1), aunque suficientes en la práctica [Grune & Jacobs 2008]). Es posible extender también el número de símbolos de preanálisis para obtener métodos y gramáticas LR(k) y LALR(k), aunque dichos métodos no son de utilidad práctica.

Una gramática LALR(1) es aquella para la que puede construirse un *autómata* LALR(1) que permita evitar los conflictos durante el análisis ascendente. Dicho autómata es un autómata reconocedor de prefijos viables para la gramática, que puede caracterizarse conceptualmente, partiendo del siguiente autómata no determinista:

- Los estados del autómata son elementos de la forma [A ::= $\alpha.\beta$, Φ], siendo A ::=$\alpha\beta$ una producción de la gramática y Φ el conjunto de símbolos terminales de preanálisis asociado a dicho elemento.

- Las transiciones son de la forma [A ::= $\alpha.X\beta$, Φ] -X-> [A ::= $\alpha X.\beta$, Φ], o bien de la forma [A ::= $\alpha.B\beta$, Φ] -λ-> [B::=.γ, Ψ], siendo Ψ el conjunto *primeros* obtenidos de todas las posibles formas sentenciales en $\beta\Phi$ (los *primeros* de una forma sentencial es el conjunto de símbolos terminales por los que comienzan las cadenas derivables desde dicha forma sentencial).

La aplicación del método de *construcción por subconjuntos* [Aho et al. 2006] a dicho autómata permite construir un autómata determinista reconocedor de prefijos viables equivalente, denominado *autómata* LR(1). Entonces, es posible identificar el *corazón* de cada estado, eliminando los componentes de preanálisis de los elementos, y fusionar los estados que albergan el mismo corazón (en dicha fusión, los símbolos de preanálisis para cada elemento en los distintos estados fusionados se unen). Existen, así mismo, algoritmos más eficientes para la construcción del autómata LALR(1), que evitan la construcción explícita del autómata LR(1) [Aho et al. 2006].

La Figura 2.2.4 muestra el autómata LALR(1) construido para la gramática de ejemplo de las expresiones aritméticas. Nótese que se introduce un axioma extendido *Expresión'* para la inicialización de los símbolos de preanálisis con el símbolo de terminación $, que comúnmente suele ser el carácter *fin de sentencia* o de *fichero*. A partir de este autómata es posible construir la *tabla de análisis sintáctico*, sin más que codificar tabularmente la función de transición de dicho autómata. Así mismo, para las entradas asociadas con terminales, será posible indicar la acción a realizar por el analizador:

- Acción de desplazamiento (*shift s*). Si existe una transición etiquetada por un terminal *a* desde un estado s_0 del autómata a otro s_1, en lugar de colocar simplemente s_1 en la entrada [s_0, a], se coloca *shift* s_1, indicando una acción de desplazamiento.

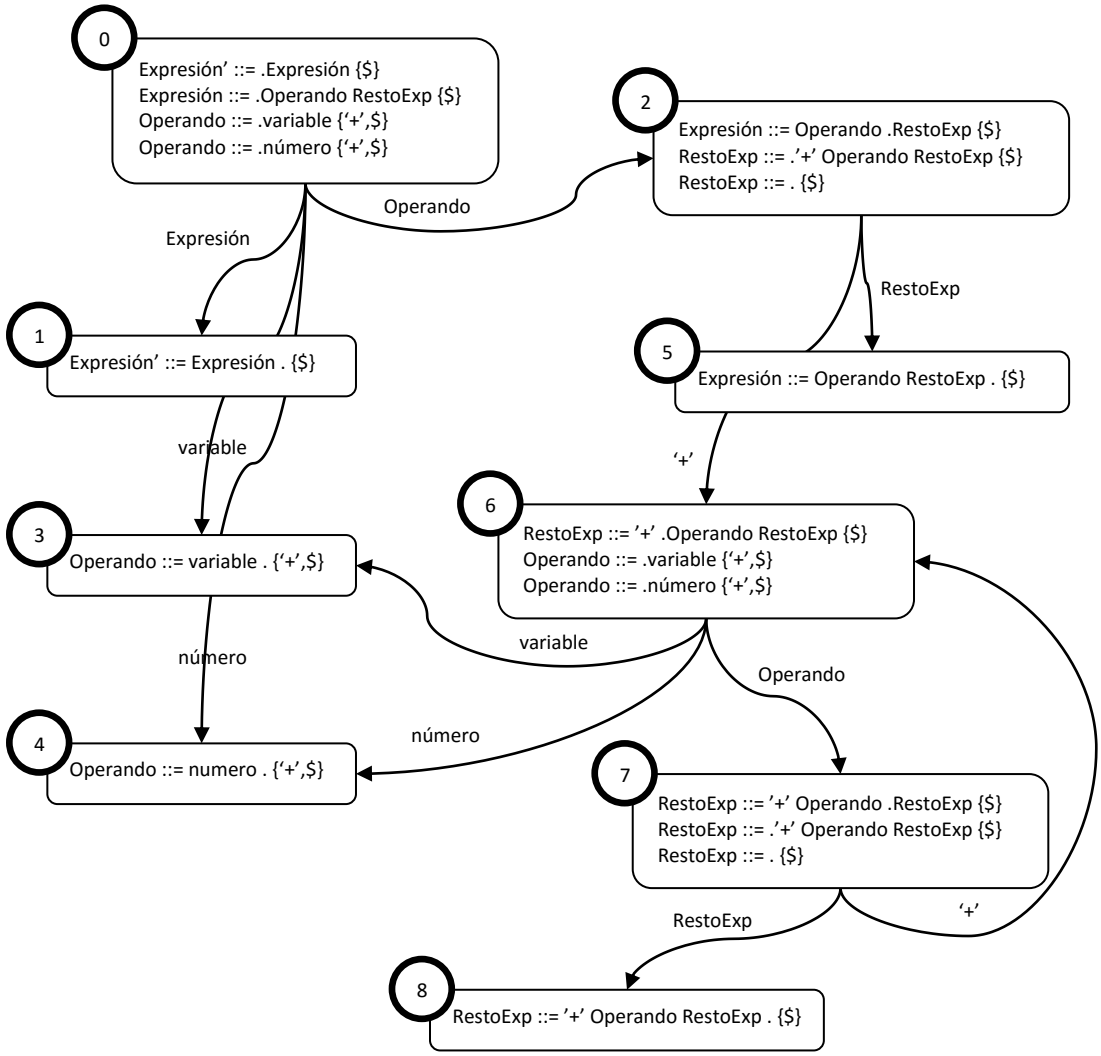


Figura 2.2.4. Autómata LALR(1) generado para la gramática de ejemplo de la Figura 2.2.1. Los elementos [A ::= α.β, {a_1,..,a_k}] se abrevian como A ::= α.β {a_1,..,a_k}.

- Acción de reducción (*reduce* A ::= α): Si en un estado *s* existe un elemento de la forma [A ::= α., Φ], para cada *a* en Φ se añade a la entrada [*s*, *a*] una acción *reduce* A ::= α, indicando una reducción por la producción A ::= α (en la práctica, en lugar de la producción se utiliza su número de orden o cualquier otra referencia que resulte conveniente).

- Acción de aceptación (*accept*). Cuando en un *estado s* hay un elemento de la forma [S'::=S., {$}], siendo S' el axioma de la gramática expandida (es decir, la gramática original con el nuevo axioma *S*' y la nueva producción *S*' ::= *S*), en lugar de una acción *reduce* S'::=S en [*s*, $] se añaden una acción de *accept*, que indica la aceptación de la sentencia de entrada.

En la Figura 2.2.5 se muestra la tabla de análisis sintáctico que se obtiene del autómata LALR(1) de la Figura 2.2.4.

Reglas Gramaticales:			1. Expresión ::= Operando RestoExp 2. RestoExp ::= '+' Operando RestoExp 3. RestoExp ::= λ 4. Operando ::= variable 5. Operando ::= número				
Estado	**número**	**variable**	**+**	**$**	**Operando**	**Expresión**	**RestoExp**
0	shift 4	shift 3			2	1	
1				accept			
2			shift 6	reduce 3			5
3			reduce 4	reduce 4			
4			reduce 5	reduce 5			
5				reduce 1			
6	shift 4	shift 3			7		
7			shift 6	reduce 3			8
8				reduce 2			

Figura 2.2.5. Tabla de análisis sintáctico LALR(1) obtenido de su autómata LALR(1) de Figura 2.2.4.

Utilizando esta tabla es posible optimizar la implementación del algoritmo de análisis sintáctico. Para ello, en la pila de análisis es posible intercalar los estados del autómata reconocedor de prefijos viables (el autómata LALR(1), en este caso), para evitar tener que reconocer completamente el prefijo viable de la pila cada vez a fin de determinar la siguiente acción a realizar. La Figura 2.2.6 ilustra el proceso de análisis resultante.

Pila de análisis sintáctico	Resto de la cadena de entrada	Entrada:Acción Desplazamiento-Reducción
[0]	"x+20"	[0,variable]:Shift 3
[0, 3:variable]	"+20"	[3,+]:Reduce 4 [0,Operando]:2
[0, 2:Operando]	"+20"	[2,+]:Shift 6
[0, 2:Operando, 6:'+']	"20"	[6,número]:Shift 4
[0, 2:Operando, 6:'+', 4:número]	""	[4,$]:Reduce 5 [6,Operando]:7
[0, 2:Operando, 6:'+', 7:Operando]	""	[7,$]:Reduce 3 [7,RestoExp]:8
[0, 2:Operando, 6:'+', 7:Operando, 8:RestoExp]	""	[8,$]:Reduce 2 [2,RestoExp]:5
[0, 2:Operando, 5:RestoExp]	""	[5,$]:Reduce 1 [0,Expresión]:1
[0, 1:Expresión]	""	[1,$]:Accept

Figura 2.2.6. Proceso de análisis ascendente de desplazamiento-reducción con tabla de análisis sintáctico LALR(1) de Figura 2.2.5.

Es interesante indicar, por último, que, aún con la información del prefijo viable presente en la pila, y utilizando los *k* siguientes símbolos de entrada como símbolos de preanálisis para precisar qué operación realizar, la aparición de conflictos mediante la aplicación de un determinado método LR acaba surgiendo si la especificación gramatical aportada no es del tipo de gramática LR soportado por dicho método, debiéndose aplicar transformaciones a dicha gramática para transformarlas a una del tipo apropiado. Esta pérdida de potencia expresiva es evitable mediante el uso de algoritmos de análisis ascendentes más avanzados, como el algoritmo GLR (ver el Capítulo 5 para más detalles) [Tomita 1986]. Este algoritmo es capaz de seguir el proceso de análisis respecto a los analizadores sintácticos específicos cuando aparecen conflictos y, por consiguiente, permitir el manejo de gramáticas generales, incluso ambiguas. Respecto a los analizadores descendentes, el conjunto de gramáticas LR(k) es menos restrictivo que el LL(k) soportado por los analizadores descendentes. No obstante, las implementaciones de los analizadores descendentes son más sencillas y directas a partir de las especificaciones gramaticales.

2.2.4 Esquemas de traducción

La finalidad de los analizadores sintácticos es fundamentalmente reconocer las sentencias que pertenecen al lenguaje descrito por una especificación gramatical. Si se desea añadir un comportamiento a medida que se analiza una sentencia, se pueden intercalar acciones que se ejecuten a medida que se reconocen los fragmentos de la sentencia de entrada. Los analizadores que poseen esta característica se denominan *traductores*. Trasladado al campo de esta tesis, el permitir acciones en contexto o dotar de semántica a los analizadores sintácticos nos permite describir el procesamiento que se desea realizar durante el análisis de un documento. Para describir esta semántica pueden utilizarse *esquemas de traducción*. Estos esquemas constan de acciones vinculadas a uno o varios elementos sintácticos de la gramática, que se ejecutarán en un momento determinado durante el análisis. Para ello, los esquemas de traducción añaden fragmentos de código al cuerpo de las producciones de la especificación gramatical y expresan el acceso al *estado local* de la producción durante el análisis. La forma exacta de ejecución de las acciones semánticas entre un analizador ascendente y uno descendente es diferente [Aho et al. 2006], aunque ambos aportan, en teoría, la misma potencia de traducción. Existen herramientas que facilitan la generación de traductores descendentes a partir de esquemas de traducción como JavaCC y ANTLR, y ascendentes como YACC y CUP. A continuación, se presentan detalles sobre estas herramientas generadoras de traductores.

2.2.4.1 Generación de traductores descendentes: JavaCC y ANTLR

Java Compiler Compiler o JavaCC [Kodaganallur 2004] es una herramienta que produce implementaciones de traductores descendentes en código Java, aportando especificaciones gramaticales con esquemas de traducción que contienen acciones escritas en Java. Estos traductores se caracterizan por ser predictivos, con soporte a gramáticas LL(k). A menor

número *k* de símbolos de preanálisis, mayor es la eficiencia de la implementación del analizador sintáctico. Es por ello que JavaCC por defecto genera implementaciones para gramáticas LL(1) y permite extensión local a k símbolos mediante anotaciones en la especificación de entrada. Una de las características de JavaCC es que incluye su propio generador de *analizadores léxicos*, es decir, un analizador de bajo nivel para el reconocimiento de tokens. Las especificaciones JavaCC tienen el aspecto de la Figura 2.2.7, donde se muestra cómo sería una especificación del traductor para la gramática de ejemplo de expresiones aritméticas que permita calcular la suma. Concretamente se distinguen tres partes:

- Configuración del gestor de tokens. JavaCC permite especificar un analizador léxico o gestor de tokens, en donde los tokens se definen mediante expresiones regulares en la sección delimitada por la palabra reservada *TOKEN*. Los caracteres que se omitirán en el análisis se delimitan con la palabra reservada *SKIP*.

- Configuración del traductor. Declaración de métodos en código Java para la inicialización y ejecución del traductor generado, delimitado por las palabras reservadas *PARSER_BEGIN* y *PARSER_END*.

- Especificación del traductor. Descripción de la gramática incontextual con posibilidad de intercalar código Java para especificar las acciones semánticas que se ejecutarán cuando el proceso de análisis alcanza los puntos en los que se han definido. Cada no terminal cabeza de producción puede asimilarse con un método Java, describiendo en su cuerpo las acciones semánticas en código Java de manera intercalada junto con la secuencia de no terminales y/o terminales. Se puede describir estos cuerpos de producción en notación EBNF. Mientras que el valor devuelto por un no terminal se puede especificar en una acción semántica adecuada, el valor devuelto por un terminal es del tipo que devuelva el analizador léxico incluido (es decir, de tipo token). Así pues, los terminales son tokens devueltos por el gestor de tokens y se declaran entre “<” “>”. Mediante la palabra reservada *LOOKAHEAD(k)* antes de un no terminal, se amplía el número de los *k* símbolos de preanálisis utilizados sólo en ese punto, lo que permite a JavaCC soportar localmente gramáticas LL(k) manteniendo el resto de la gramática como LL(1) de una manera eficiente.

ANother Tool for Language Recognition o ANTLR [Parr & Quong 1995] es un generador de traductores descendentes similar a JavaCC, aunque presenta funciones adicionales a JavaCC. Aparte de permitir generar traductores no sólo en código Java, sino también en un amplio abanico de lenguajes de programación, y de posibilitar la generación automática de árboles de análisis sintácticos abstractos incluyendo herramientas para su visualización y recorrido, la principal característica de ANTLR es permitir un número ilimitado y adaptativo de símbolos de preanálisis para cada producción. Para ello, las cadenas predictoras para cada producción se substituyen por autómatas finitos deterministas predictores, que deciden cuándo es posible aplicar cada producción. El método empleado por ANTLR para generar este tipo de analizadores se denomina LL(*) [Parr & Fisher 2011]. Por lo demás, las especificaciones ANTLR son muy similares a las de JavaCC, excepto en sus versiones más evolucionadas, que cada vez adoptan mayor simplicidad sintáctica.

```
options {LOOKAHEAD=1;} //establecer ll(k) a ll(1)

/* configuración del traductor */
PARSER_BEGIN(Traductor)
  //declaración de importaciones de clases y código adicional
  import java.util.Map;
  import java.io.InputStream;

  public class Traductor {
    private Map<String,Integer> variables;
    public static int run(InputStream stream, Map<String,Integer> variables) throws ParseException {
      Traductor t = new Traductor(stream);
      this.variables = variables;
      return t.Expresion();
    }
    private static int suma(int a, int b) { return a + b; }
    private static int valorDe(String nombre) { return variables.get(nombre); }
  }
PARSER_END(Traductor)

/* configuración del gestor de tokens */
SKIP: {" "|"\t"|"\f"|"\r"}
TOKEN :
{
  <mas: "+"> | <numero: (<digito>)+> | <variable: (<letra>(<caracter>)*)> |
  //reglas de composicion de tokens
  <#digito: ["0"-"9"]> | <#letra: (["a"-"z"] | ["A"-"Z"])> | <#caracter: (<letra>|<digito>)>
}

/* especificación del traductor */
int Expresion(): {int a,b;} { a=Operando() b=RestoExp() <EOF> {return suma(a,b);} }
int RestoExp(): {int a,b;} { <mas> a=Operando() b=RestoExp() {return suma(a,b);} | {return 0;} }
int Operando(): {Token tok;} { tok=<variable> {return valorDe(tok.image);} | tok=<numero> {return Integer.parseInt(tok.image);} }
```

Figura 2.2.7. Especificación JavaCC del traductor para la gramática de ejemplo.

2.2.4.2 Traductores Ascendentes: CUP y YACC

CUP (Constructor of Useful Parsers) [Hudson 1999] es una herramienta generadora de traductores ascendentes en código Java, soportando especificaciones gramaticales mediante esquemas de traducción. CUP se presenta como una versión evolucionada para Java de YACC (Yet Another Compiler Compiler) [Johnson 1975], el generador de traductores ascendentes de tipo desplazamiento-reducción en código C para gramáticas de tipo LALR(1). Estos generadores no incluyen su propio analizador léxico, sugiriendo el uso de generadores de analizadores léxicos complementarios como LEX [Schreiner & Friedman 1985] para C, o jFLex para Java [Appel 1997].

La Figura 2.2.8 muestra una especificación en CUP del traductor para la gramática de ejemplo de las expresiones aritméticas que calcula la suma. Puede observarse tres partes diferenciadas:

- Configuración del traductor. Permite incluir cualquier método o fragmento de código Java para la ejecución del traductor y su inicialización. También permite incluir código a ejecutar antes de obtener un token o después de haber sido éste obtenido por el analizador léxico.

```
import java_cup.runtime.*;
import java.util.Map;
import java.io.*;

/* configuración del traductor */
parser code {:
  protected Map<String,Integer> variables;
  public static int run(Scanner analizadorLexico, Map<String,Integer> variables) throws Exception {
    parser p = new parser();
    p.variables = variables;
    p.setScanner(analizadorLexico);
    Symbol result = p.parse();
    return (int)result.value;
  }
  protected static int suma(int a, int b) { return a + b; }
  protected int valorDe(String nombre) { return variables.get(nombre); }
:};

/* declaración de terminales y/o no terminales con o sin tipo */
terminal mas, END;
terminal String variable; terminal int numero;
non terminal Integer Expresion, Operando, RestoExp;

/* especificación del traductor */
Expresion ::= Operando:a RestoExp:b END {: RESULT = parser.suma(a,b); :};
RestoExp ::= mas Operando:a RestoExp:b {: RESULT = parser.suma(a,b); :} | {: RESULT = 0; :};
Operando ::= variable:a {: RESULT = parser.valorDe(a); :} | numero:b {: RESULT = b; :};
```

Figura 2.2.8. Especificación CUP del traductor para la gramática de ejemplo.

- Declaración de terminales/no terminales. Puesto que el analizador léxico debe ser aportado externamente, es necesario declarar qué elementos de la gramática son símbolos terminales y cuáles son símbolos no terminales, permitiendo indicar el tipo devuelto tras realizarse una reducción por un símbolo no terminal, y el tipo devuelto tras realizarse un desplazamiento por un símbolo terminal.

- Especificación del traductor. Las producciones se describen en BNF, permitiendo asignar variables tras el símbolo ":" a terminales o no terminales como una forma más practica respecto a YACC de acceso a los registros semánticos de dichos elementos sintácticos. Estos registros guardan la información semántica asociada a un terminal o no terminal concreto. Al ser el traductor generado de tipo desplazamiento-reducción, la información semántica queda retenida en la pila de análisis sintáctico, cuyo acceso a sus elementos se especifica en CUP mediante las variables asociadas manualmente a los elementos del cuerpo de las producciones. Por último, las acciones semánticas se añaden al final de cada producción y se ejecutan cuando se reducen dichas producciones.

2.2.5 Gramáticas de Atributos

Las *gramáticas de atributos* constituyen una extensión de las gramáticas incontextuales mediante la adición de semántica. Este formalismo, que se detalla en la sección 2.2.5.1, permite elevar el nivel descriptivo del procesamiento mediante el enriquecimiento de las reglas de producción con atributos y ecuaciones semánticas. Esto se traduce en un nivel declarativo superior respecto a los esquemas de traducción, pues la potencia declarativa de este formalismo permite establecer una forma de cómputo más avanzada, que permite obviar el orden en el que los valores de los atributos se determinan, como se expone en la sección 2.2.5.2, y facilita también la modularización de las especificaciones mediante diferentes técnicas, como las que se presentan en la sección 2.2.5.3.

2.2.5.1 El formalismo descriptivo

Las gramáticas de atributos son un formalismo propuesto por el matemático norteamericano Donald E. Knuth [Knuth 1968] para la especificación y la caracterización de la *semántica* de los lenguajes incontextuales. Su grado de abstracción permite la descripción declarativa de traductores, a un nivel mucho más alto que el de los esquemas de traducción. Así mismo, también permiten la generación automática de procesadores eficientes para el procesamiento de lenguajes. La notación que adopta se estructura en términos de las gramáticas incontextuales, al ser el formalismo una extensión de las mismas, y la descripción de la semántica se realiza mediante:

- *Atributos semánticos*. El análisis sintáctico de una sentencia, visto anteriormente, induce un árbol de análisis sintáctico asociado. Los atributos semánticos permiten asociar valores a los nodos de dicho árbol en términos de pares *atributo* - *valor*. Los atributos semánticos pueden ser de dos tipos: *atributos heredados*, que representan información de contexto que se propaga desde el nodo padre o desde los nodos hermanos, y *atributos sintetizados*, que representan los significados y cuyo valor se establece en función de atributos heredados en el nodo y/o de los atributos sintetizados de los nodos hijo. Los atributos semánticos quedan asociados a un terminal o no terminal concreto, lo que implica una asociación implícita a sus correspondientes nodos en el árbol de análisis. En las gramáticas de atributos, ambos tipos de atributos se refieren mediante el símbolo terminal o no terminal asociado, seguido opcionalmente de un subíndice, seguido de un punto, y, por último, el nombre del atributo. La presencia de subíndice sirve para referirse unívocamente a los terminales y/o no terminales tanto de la cabeza como del cuerpo de la producción, estableciendo el subíndice 0 para la primera ocurrencia del símbolo e incrementándose en 1 para cada ocurrencia sucesiva.

- *Ecuaciones semánticas*. Expresan la forma de computar los valores asociados a los atributos semánticos en función de otros atributos, operaciones denominadas *funciones semánticas*, o valores literales, y permiten establecer la información de sus

atributos heredados. Las ecuaciones semánticas quedan asociadas a cada producción, ya que establecen cómo computar los valores de los atributos sintetizados de la cabeza y de los atributos heredados pertenecientes a los símbolos del cuerpo de dicha producción. La sintaxis utilizada consiste en el atributo semántico al que se le asigna el valor de la ecuación, seguido del símbolo "=", seguido de la expresión con las operaciones y/o valores que definen dicha ecuación.

Los símbolos terminales pueden tener *atributos léxicos*, que no son más que atributos sintetizados cuyo valor se establece de manera externa durante el análisis léxico de la sentencia de entrada. Concretamente, dicho valor se obtiene a través del valor que aloja el token que se corresponde a dicho terminal. Por otro lado, el axioma también puede tener atributos heredados mediante un correcto establecimiento de su valor.

Expresión ::= Operando RestoExp
  Expresion.valor = *suma*(Operando.valor, RestoExp.valor)
  Operando.variables_h = Expresión.variables_h
  RestoExp.variables_h = Expresión.variables_h

RestoExp ::= '+' Operando RestoExp
  $RestoExp_0$.valor = *suma*(Operando.valor, $RestoExp_1$.valor)
  $RestoExp_1$.variables_h = $RestoExp_0$.variables_h
  Operando.variables_h = $RestoExp_0$.variables_h

RestoExp ::= λ
  RestoExp.valor = "0"

Operando ::= variable
  Operando.valor = *valorDe*(variable.nombre, Operando.variables_h)

Operando ::= número
  Operando.valor = número.valor

Figura 2.2.9. Ejemplo de gramática de atributos para las expresiones que calculan la suma.

En la Figura 2.2.9 se muestra la gramática de ejemplo de las expresiones aritméticas extendida con semántica, convirtiéndose en una gramática de atributos donde las funciones semánticas escritas en cursiva aluden a implementaciones específicas de las correspondientes operaciones. Como puede observarse, existe un atributo heredado *variables_h* que permite propagar una tabla con el valor asociado a las variables, y un atributo sintetizado *valor* sobre el que se propaga el resultado que se va calculando con la función semántica de *suma*.

2.2.5.2 Evaluación semántica

Un aspecto muy importante a tener en cuenta en una gramática de atributos es el orden de cómputo de las ecuaciones semánticas. Este orden es implícito, debido a que puede inferirse en función de las dependencias entre los atributos semánticos en el árbol de análisis sintáctico, ahora atribuido. Efectivamente, únicamente debe garantizarse que, antes de computar el valor de un atributo, se hayan computado todos los valores de los atributos de los que éste

depende. Estas dependencias quedan reflejadas implícitamente en el grafo de dependencias subyacente al árbol de análisis sintáctico atribuido. Esto dota al formalismo de un nivel descriptivo mayor que el de los esquemas de traducción típicamente empleados en los procesadores de lenguaje para dotar de semántica a las gramáticas incontextuales, como JavaCC, YACC, CUP, etc. descritos en la sección 2.2.4, donde sí hay que especificar explícitamente un orden de evaluación [Aho et al. 2006]. En la Figura 2.2.10 se muestra el grafo de dependencias de atributos semánticos impuesto por la gramática de atributos de la Figura 2.2.9 tras el análisis de una sentencia de entrada.

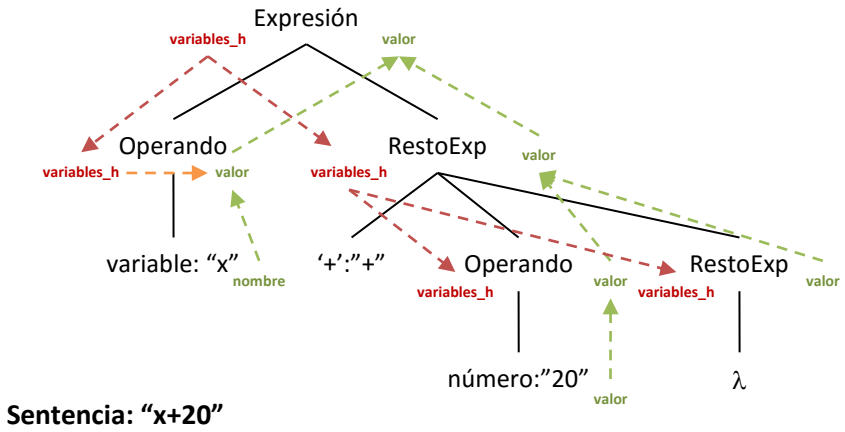


Figura 2.2.10. Grafo de dependencias para la sentencia "x+20" impuesto por la gramática de atributos de la Figura 2.2.9.

El orden de evaluación en las gramáticas de atributos puede determinarse mediante *métodos estáticos* o *dinámicos* [Alblas 1991] siempre y cuando los grafos de dependencias sean no circulares, es decir, sin presencia de ciclos. Los métodos estáticos determinan un orden de evaluación apto para cualquier sentencia de entrada tras realizar un análisis de la gramática de atributos, mientras que en los métodos dinámicos el orden queda determinado por la propia sentencia de entrada que se analiza. De esta forma:

- Si bien los métodos estáticos son computacionalmente más eficientes, el ámbito de gramáticas que soportan queda restringido a tipos particulares de gramáticas de atributos, como son las gramáticas fuertemente no circulares [Jourdan 1984], o las gramáticas de atributos ordenadas [Kastens 1980]. Cabe destacar, así mismo, que existen otros métodos específicos de evaluación para los métodos estáticos de manera predeterminada, que aprovechan las características de unas clases de gramáticas de atributos muy específicas para realizar una evaluación muy eficiente, incluso mientras se realiza el análisis sintáctico de la sentencia de entrada [Akker et al. 1991, Wilhelm 1982].

- Por el contrario, los métodos dinámicos permiten el manejo de gramáticas de atributos no circulares arbitrarias [Cohen & Harry 1979] mediante el establecimiento de un orden

topológico de los atributos en el grafo de dependencias asociado, impuesto por la descripción gramatical. En cuanto a la implementación de estos métodos dinámicos, la evaluación puede realizarse de manera *dirigida por los datos* o bien *bajo demanda*. La primera manera consiste en el cómputo de todos los valores de atributos en el árbol de análisis sintáctico atribuido tan pronto como estén sus dependencias resueltas [Kennedy & Ramanathan 1979], mientras que la segunda consiste en el cómputo del valor de aquellos atributos que son requeridos evitando el cómputo de atributos innecesarios [Jalili 1983, Magnusson & Hedin 2007].

Por último, existen diversas herramientas con soporte a las gramáticas de atributos como GAG [Kastens et al. 1994], FNC-2 [Jourdan & Parigot 1991], o Silver [Wyk et al. 2010], cuyo ámbito de gramáticas soportado queda restringido al método de evaluación aplicado que, por lo general, es de tipo estático.

2.2.5.3 Técnicas de modularización

Las propuestas de modularización de gramáticas de atributos surgen con el fin de incrementar el mantenimiento y facilitar la elaboración de dichas especificaciones. Atienden a un nivel conceptual más elevado que el formalismo básico, adoptando principios de modularización como los aplicados a los lenguajes de programación. De hecho, el formalismo de las gramáticas de atributos posee, por su naturaleza, un carácter modular [Farrow et al. 1992], debido, sobre todo, al carácter implícito del orden de evaluación, que puede reorganizarse conforme se añaden más y más aspectos a la especificación.

En relación con las técnicas de modularización de gramáticas de atributos, en [Paakki 1995] se realiza una clasificación en función de concebir los siguientes aspectos como *módulo*:

- No terminales. El módulo queda caracterizado mediante el sublenguaje definido bajo un no terminal, permitiendo su añadido a otros lenguajes mediante la introducción de dicho no terminal en sus especificaciones. La semántica del sublenguaje definido por el no terminal como módulo añadido queda oculta, permaneciendo únicamente visible los atributos de dicho no terminal.

- Atributos semánticos. El módulo se caracteriza como todas las reglas semánticas asociadas a un atributo semántico, lo que en su añadido a otra especificación gramatical supone añadir todas las producciones de los símbolos no terminales que poseen dicho atributo.

- Aspectos semánticos. La caracterización del módulo engloba a todas las producciones y atributos semánticos que intervienen en la especificación de un determinado aspecto semántico.

En [Kastens & Waite 1992] se propone utilizar *patrones de atribución* para facilitar el desarrollo modular de especificaciones basadas en gramáticas de atributos. Estas técnicas se centran en la simplificación de la especificación semántica de una gramática de atributos a

base de aplicar patrones que abstraen comportamientos recurrentes, facilitando su reutilización en otras especificaciones. Estos patrones pueden ser de diversos tipos:

- De reglas semánticas asociadas a símbolos gramaticales. Su aplicabilidad es efectiva cuando las ecuaciones semánticas dependen exclusivamente de atributos de un símbolo concreto de la gramática, estableciendo una asociación a nivel de símbolo-regla semántica frente a símbolo-producción, lo que permite que dichas ecuaciones o reglas semánticas sean aplicables en los árboles sintácticos en todas las apariciones de dicho símbolo.

- De herencia. Sucede en situaciones en donde ciertas ecuaciones semánticas son aplicables a un conjunto de símbolos de la gramática. Este conjunto puede clasificarse en una clase determinada y así describir la aplicación de dichas ecuaciones directamente a una clase, lo que implica su aplicación a todos los elementos del conjunto que representa, bajo el concepto de símbolos herederos del símbolo representante de la clase.

- De acceso a atributos remotos. El patrón es aplicable cuando el cómputo de un atributo perteneciente a un nodo en el árbol sintáctico depende del valor de un atributo de otro nodo diferente, denominado atributo *remoto* en el patrón. Estos atributos remotos pueden clasificarse en tres tipos según la ubicación del nodo remoto respecto al considerado: en un nodo hermano (mismo nivel), en un nodo ancestro (nodo padre o nivel superior), o en un nodo descendiente (nodo hijo o nivel inferior). Las especificaciones se vuelven más escuetas y precisas al introducir formas de expresar las dependencias entre atributos de nodos en estas situaciones no locales. Por otro lado, la reutilización de los módulos en otras especificaciones se vuelve más fácil al abstraerse las particularidades estructurales que presentan los subárboles de los atributos considerados respecto de los atributos remotos.

- De atribución acumulativa. Se aplica en función de los diversos procesamientos que se especifican en la gramática de atributos, atendiendo a la separación de dichos procesamientos en especificaciones distintas o *fragmentos de gramáticas de atributos*. De esta manera se realiza una descomposición física de las especificaciones en fragmentos más simples, a los que se pueden aplicar los anteriores patrones dando lugar a módulos reutilizables. La composición de todos estos módulos, mediante la unión adecuada de sus producciones y ecuaciones semánticas, generará una única especificación, análoga a la que se elaboraría para realizar todos los procesamientos considerados conjuntamente.

En [Farrow et al. 1992] se propone una técnica de modularidad semántica del formalismo de las gramáticas de atributos denominada *gramáticas de atributos componibles* o *fusionables*. Esta propuesta destaca sobre otras de la misma índole [Adams 1991, Dueck & Cormack 1990, Ganzinger & Giegerich 1984, Hedin 1989, Hedin 1999, Saraiva & Swiestra 1999, Vogt et al. 1989] por su mayor usabilidad en la práctica. Estas gramáticas de atributos tienen la propiedad de construirse mediante la unión de otras gramáticas de atributos más simples, denominadas

gramáticas componente, que resuelven un problema específico. Estas gramáticas se pueden componer en una sola beneficiándose de todas las características que nos aporta la especificación modular. En concreto, el formalismo además de permitir la abstracción y ocultación de los detalles de implementación a la resolución de cada problema específico, permite la reusabilidad de las gramáticas componente al fomentar una estructura jerárquica a base de composiciones incrementales con estas gramáticas. Para ello, las gramáticas componente son gramáticas de atributos que permiten a los símbolos terminales poseer atributos heredados. Con esta ampliación, estas gramáticas componentes son capaces de unirse entre sí mediante una gramática *pegamento* o constructora. Esta gramática considera eventualmente terminales en los componentes como no terminales, introduce nuevas reglas y ecuaciones semánticas que los relaciona con los axiomas de otros componentes, y así sucesivamente hasta dar lugar a la especificación global de lenguaje.

2.3 Documentos XML y su procesamiento

Esta tesis propone un nuevo enfoque para el procesamiento de documentos XML. De esta forma, esta sección presenta los conceptos más relevantes de XML de cara a la misma. La sección 2.3.1 introduce el lenguaje. La sección 2.3.2 introduce el concepto de *gramática documental*, como mecanismo para la definición de tipos particulares de documentos XML. La sección 2.3.3 y la sección 2.3.4, por último, revisan las técnicas convencionales al procesamiento de este tipo de documentos.

2.3.1 XML

Extensible Markup Language (XML) [Bray et al. 2008] es un lenguaje de *marcado* [Coombs et al. 1987] para la codificación de la información, utilizado para la estructuración y presentación del contenido de los documentos electrónicos. La Figura 2.3.1 muestra un ejemplo de documento XML. XML surge como un estándar abierto desarrollado por el World Wide Web Consortium (W3C), basado en el Standard Generalized Markup Language (SGML) [Goldfarb 1991], enfocado a un uso más sencillo mediante la eliminación de la problemática y características más específicas de este último. Debido a su sencillo formato, alta legibilidad, facilidad, fiabilidad y seguridad para compartir la información entre diversos sistemas, se ha convertido en un estándar altamente extendido por todo el mundo. La estructura de un documento XML se compone de tres partes:

- Encabezado del documento compuesto por metainformación dirigida al procesador del documento XML. Contiene la versión XML y el conjunto de caracteres utilizado. XML obliga a los sistemas de procesamiento a soportar, como mínimo, Unicode UTF-8 y UTF-16 en sus codificaciones [Bradley 2001].

- Estructura del documento mediante su especificación con una Document Type Definition (DTD). Esta especificación es importante para que el procesador de documentos XML sea capaz de comprobar y validar los documentos XML, así

como garantizar su correcto procesamiento posterior. Sin embargo, esta definición es opcional.

- Instancia documental con la información del documento estructurada mediante etiquetas de la forma especificada en la DTD.

Los lenguajes de marcado se caracterizan por el uso de etiquetas o marcas, meta-información añadida al documento, para dotar de estructura a los contenidos del documento o para explicitar otros aspectos, como puede ser la presentación de los mismos. La información del documento en XML concretamente se contiene entre pares de etiquetas o *elementos XML*: etiqueta de apertura (p.e. <etiqueta>) y etiqueta de cierre (p.e. </etiqueta>). Estos elementos pueden contener *atributos*, asociando a los pares de etiquetas una meta-información adicional de carácter *atributo-valor* (p.e. el atributo *tipo* de la etiqueta de apertura <Forma> en la Figura 2.3.1). Asimismo, estas etiquetas son de carácter *descriptivo* al establecer explícitamente la estructura de los documentos, pero no la forma de procesarlos.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE Formas SYSTEM "Formas.dtd">

<Formas>
  <Forma tipo='regular'>
    <Nombre>Cuadrado 10px</Nombre>
    <Puntos>
      <Punto2D>
        <CoordenadaX>10</CoordenadaX>
        <CoordenadaY>10</CoordenadaY>
      </Punto2D>
      <Punto2D>
        <CoordenadaX>20</CoordenadaX>
        <CoordenadaY>10</CoordenadaY>
      </Punto2D>
      <Punto2D>
        <CoordenadaX>20</CoordenadaX>
        <CoordenadaY>20</CoordenadaY>
      </Punto2D>
      <Punto2D>
        <CoordenadaX>10</CoordenadaX>
        <CoordenadaY>20</CoordenadaY>
      </Punto2D>
    </Puntos>
  </Forma>
</Formas>
```

Figura 2.3.1. Documento XML de ejemplo.

2.3.2 Gramáticas documentales

La estructura que siguen los documentos XML es jerárquica, donde existe un elemento raíz que da lugar al anidamiento de otros elementos. De esta forma, es esencial disponer de un mecanismo preciso y claro para especificar la estructura de los documentos, como la potencia que una definición gramatical es capaz de aportar para los lenguajes de marcado descriptivo [Goldfarb 1981]. Para ello es posible utilizar una *gramática documental*.

Las DTDs son un tipo simple, pero efectivo, de gramáticas documentales incluidas directamente como sublenguaje de XML. Efectivamente, la DTD aporta una definición gramatical de lenguaje de marcado que permite describir esta estructura anidada mediante expresiones regulares o *modelos de contenidos*. Sin la aportación de una DTD al documento u otro mecanismo de definición gramatical [Murata et al. 2005], las instancias documentales son igual de válidas y procesables siempre y cuando cumplan con todas las definiciones de formato requeridas por el procesador de documentos XML. Sin embargo, la DTD permite delimitar las construcciones de marcas que pueden emplearse en las instancias XML mediante un enfoque lingüístico de gramáticas formales.

```
<?xml version="1.0" encoding="UTF-8" ?>

<!ENTITY % Min3Puntos '(Punto2D,Punto2D+,Punto2D)'>

<!ELEMENT Formas (Forma)*>
<!ELEMENT Forma (Nombre, Puntos)> <!ATTLIST Forma tipo CDATA #IMPLIED>
<!ELEMENT Nombre (#PCDATA)>
<!ELEMENT Puntos (%Min3Puntos;)>
<!ELEMENT Punto2D (CoordenadaX, CoordenadaY)>
<!ELEMENT CoordenadaX (#PCDATA)>
<!ELEMENT CoordenadaY (#PCDATA)>
```

Figura 2.3.2. DTD del documento XML de ejemplo (Figura 2.3.1).

En una DTD, cada uno de los tipos de elemento se declara mediante !ELEMENT, especificado una expresión regular o *modelo de contenidos* asociado al tipo de elemento, que describe cómo se estructuran los elementos que contiene. De esta forma, es posible establecer restricciones en el tipo de elementos y atributos que podrán contemplar las instancias documentales. Estas expresiones se forman en función de elementos XML (etiquetas), de subexpresiones a través de entidades mediante !ENTITY (las subexpresiones se emplean añadiendo el prefijo "%" y el sufijo ";" a su nombre), de contenido textual mediante #PCDATA, de contenido vacío mediante EMPTY, o incluso mediante ANY para expresar cualquier estructura de subelementos sin limitación. Los atributo-valor asociados a los elementos se declaran mediante !ATTLIST como una lista, que permite indicar el tipo de cada atributo (tipo caracteres mediante CDATA, identificador único mediante ID, referencia a un elemento con IDREF, etc.), y el estado de su valor (valor por defecto si no se declara el atributo en el documento XML, valor requerido mediante #REQUIRED, opcional con #IMPLIED, etc.).

La Figura 2.3.2 muestra un ejemplo de DTD para una instancia documental como la presentada en Figura 2.3.1. En ella Formas es el elemento raíz, donde Forma se anida cero o más veces dentro del elemento anterior. A su vez, los elementos Nombre y Puntos se anidan dentro de Forma. Para el elemento Forma, se declara como contenido (Nombre, Puntos) para declarar que se anidan consecutivamente los elementos Nombre y Puntos. Por su parte, el elemento Nombre contendrá exclusivamente un contenido textual al estar formada su expresión regular únicamente por #PCDATA.

Aunque las DTDs se presentan como las gramáticas documentales para XML más simples y de gran uso práctico, presentan también de una serie de limitaciones [Megginson et al. 1998].

Estas limitaciones atienden al poder expresivo limitado de tipado de los contenidos y atributos [Birceck et al. 2001] que imposibilita la realización de la comprobación del tipo de estos a nivel de validación de instancia documental mediante la DTD, a la modularidad basada en entidades de tipo parámetro [Megginson et al. 1998] que dificulta la formación de DTDs a partir de la combinación o extensión de otras, o a la simple dificultad de expresar el desorden de los subelementos de los que se compone un elemento XML. Las alternativas de gramáticas documentales [Jelliffe 2001, Lee et al. 2000, Murata et al. 2001, Murata et al. 2005, Vlist 2001]. surgen para hacer frente a las limitaciones que presentan las DTDs. De entre estas alternativas, destacan distintos *lenguajes de esquema* XML, que se expresan empleando la misma sintaxis de XML, y entre los que cabe resaltar:

- RELAX NG [Clark et al. 2001-a, Clark et al. 2001-b]. Es fruto de la mezcla de los lenguajes de esquema RELAX [Murata 2000] y TREX [Clark 2001-a, Clark 2001-b]. El primero posee un fundamento basado en la teoría de los lenguajes de árboles regulares [Thatcher 1967, Thatcher 1973, Takahashi 1975, Comon et al. 1997, Murata 1995, Murata 1999], con una orientación gramatical, mientras que el segundo se centra en la aplicación de las expresiones regulares sobre árboles para descripciones con orientación a tipos. Las especificaciones de RELAX NG se presentan como un lenguaje sencillo frente a otras alternativas, como XML Schema.

- XML Schema [Fallside 2001, Thompson et al. 2001, Biron et al. 2001]. Es un lenguaje de esquemas recomendado por el W3C, que destaca por la formulación incremental de tipos de datos para la estructuración de los documentos. El lenguaje incluye nuevos tipos predefinidos y a su vez permite establecer nuevos tipos simples o complejos, en función de las restricciones que se impongan. Las especificaciones resultantes son muy potentes, pero también bastante complejas y poco legibles.

- Schematron [Jelliffe 2002]. Es un lenguaje que se centra en la delimitación de las formas estructurales que el lenguaje de marcado puede presentar, cuyo fundamento reside en las gramáticas de asertos [Raggett 1999], y que puede combinarse con otros lenguajes de esquema para incrementar su potencia expresiva.

2.3.3 Marcos genéricos de procesamiento de documentos XML

El marcado descriptivo XML permite codificar la información de una forma estructurada y sencilla [Coombs et al. 1987], pero no establece ninguna norma para procesar dicha información. Si nuestro objetivo es realizar tareas complejas sobre estos documentos, es necesario utilizar un marco de procesamiento XML. Los marcos más genéricos [Lam et al. 2008] abarcan las funcionalidades básicas y necesarias para satisfacer las necesidades de procesamiento de documentos XML, como son, en resumen, funcionalidades de lectura, comprobación de formato y validación de documentos XML, así como la disposición de los datos leídos para su procesamiento. Como tipos de marcos de procesamiento genérico XML más utilizados destacan los *marcos orientados a árboles* y *marcos orientados a eventos.*

2.3.3.1 Marcos orientados a árboles

Los marcos orientados a árboles se caracterizan por organizar los datos leídos de un documento XML mediante una estructura arborescente. La característica fundamental de estos marcos reside en la capacidad de acceso en cualquier momento a cualquier dato contenido en el documento XML, lo que permite realizar operaciones complejas, de búsqueda, recursivas y reiteradas sobre los datos. Sin embargo, toda la información y estructura arborescente se almacena en memoria durante el procesamiento, lo que, para documentos muy grandes, implica un elevado consumo de memoria y tiempo de construcción de la estructura.

Como un estándar de marco de procesamiento orientado a árboles puede destacarse Document Object Model (DOM) [DOM 2009], fruto del W3C. Su aporte reside en una interfaz estándar junto a un modelo estándar para la construcción de la estructura arborescente y representación de los documentos XML, así como una interfaz estándar para la modificación y manipulación de dichos objetos y su contenido. Su uso está enfocado a lenguajes orientados a objetos, sin establecer una representación física de cómo deben ser las estructuras arborescentes, sino únicamente las interfaces que permiten el acceso y la manipulación de dichas estructuras.

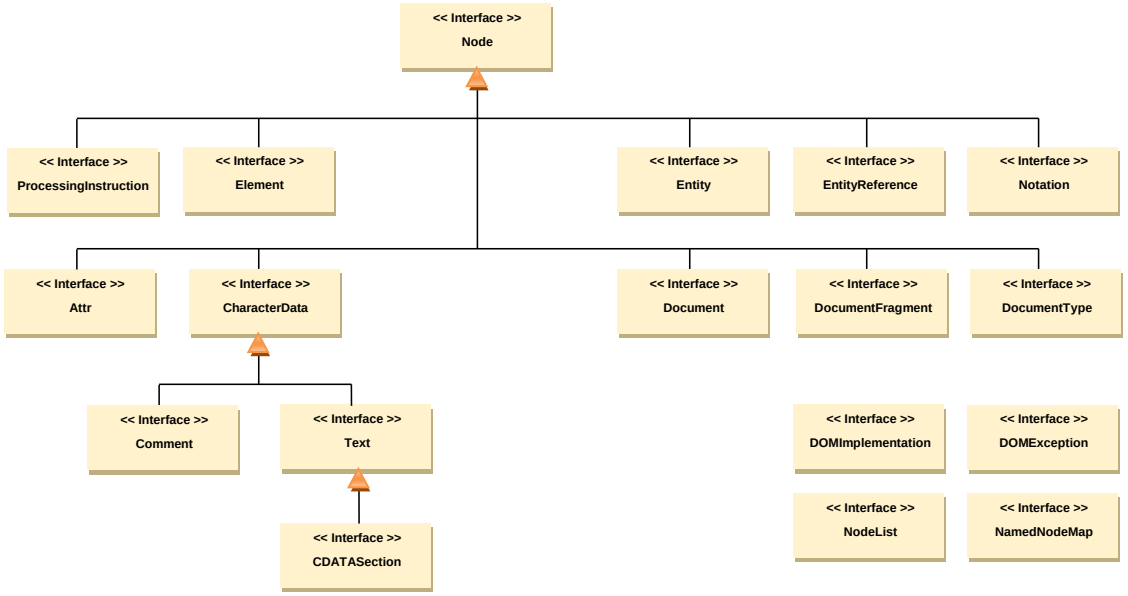


Figura 2.3.3. Diagrama de clases de la estructura DOM de nivel 1.

Existen varios niveles de arquitectura DOM o especificaciones que introducen distintas características. La Figura 2.3.3 muestra la arquitectura DOM, de nivel 1. En este nivel 1, la estructura arborescente contempla nodos cuyo tipo depende de su contenido, tales como *Text* (su valor es el contenido del texto) o *Element* (una secuencia de nodos hijos y una tabla de atributos), etc. Mediante métodos factoría y la interfaz *Document*, se crean los diferentes

nodos y se permite realizar implementaciones concretas y especializadas sobre el resto de elementos. Con todo ello, estas interfaces permiten recorrer la estructura tanto de manera iterativa sobre los nodos, o utilizando las relaciones que se establecen entre ellos.

2.3.3.2 Marcos orientados a eventos

Los marcos orientados a eventos se caracterizan por la lectura secuencial del documento disparando un evento cada vez que se reconoce un componente. Sólo es necesario implementar qué operación debe realizarse tras reconocer un componente concreto. Esto los convierte en rápidos en procesamiento y sencillos de programar. En contraste con los marcos orientados a árboles, al no crear ni mantener ninguna estructura en memoria, el uso de recursos no crece respecto al tamaño de los documentos procesados, pero por otro lado dificulta la realización de operaciones complejas, como operaciones reiteradas y recursivas, sobre el contenido del documento. Como ejemplos de marcos de procesamiento orientados a eventos destacables cabe citar a SAX y StAX:

- SAX o Simple API for XML [Brownell 2002] se caracteriza por un conjunto de interfaces estándar para la lectura de los componentes de los documentos XML y para el tratamiento de los eventos que estos producen. El marco se caracteriza por poseer un mecanismo de control de tipo *push*, en donde el control reside en el marco en sí y los eventos se disparan a medida que se realiza el análisis del documento XML. A través de los manejadores de eventos de SAX, es posible extender la funcionalidad de un procesador XML en función de los parámetros e implementación de los métodos manejadores de los eventos que se disparan. En conformidad con SAX, la construcción de procesadores XML puede realizarse de manera modular, permitiendo el tratamiento y la notificación de eventos de una manera multinivel en función de la capa de procesamiento sobre la que se construya cada procesador. De esta manera, un primer nivel puede centrarse exclusivamente en el análisis del documento, y en otro nivel, realizarse tareas concretas de procesamiento. Esta es una de las ventajas heredadas del enfoque *push* [Dabek et al. 2002], aunque como contrapartida, posee la desventaja de requerir el mantenimiento del contexto de procesamiento entre las invocaciones de los eventos a fin de garantizar un correcto control sobre el proceso llevado a cabo. La Figura 2.3.4 presenta las interfaces de gestión de eventos de SAX. Entre ellas, la interfaz *DocumentHandler* define métodos que se invocan tras el reconocimiento de un componente del documento XML. Para aspectos de lectura, se dispone de los métodos proporcionados por las interfaces *XMLReader* y *XMLFilter*. Para otros aspectos, tales como los relativos a la DTD o detección de errores, se dispone de las interfaces *DTDHandler* y *ErrorHandler*. Por último, la clase *DefaultHandler* proporciona una implementación básica de partida, que puede extenderse para concretar una implementación de los métodos de tratamiento de los eventos.

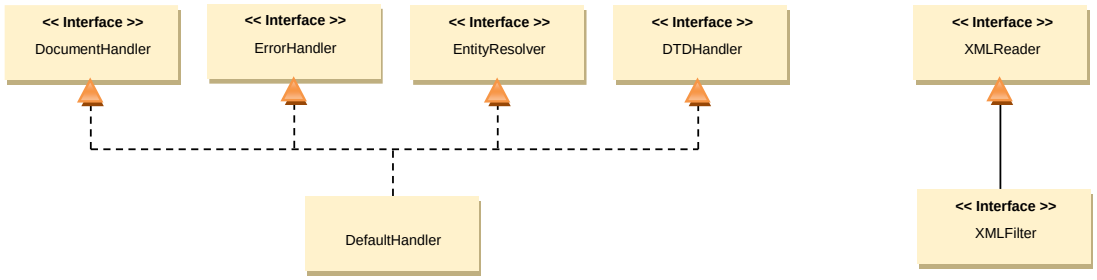


Figura 2.3.4. Interfaces de gestión de eventos SAX.

- StAX o Streaming API for XML [McLaughlin 2006] es un conjunto de interfaces para la lectura de los componentes de los documentos XML adoptando un mecanismo de control de tipo *pull*. Este enfoque es el inverso al enfoque *push* de SAX, ya que el control reside en la propia lógica de la aplicación, en lugar de en el marco De esta forma, el proceso de análisis del documento es tan sencillo como el que se utiliza tradicionalmente en los traductores predictivos descendentes analizados anteriormente. Así mismo, adquiere como desventaja y ventaja las contrarias a las de un marco *push*: la extensibilidad se complica, pero el mantenimiento de contexto ya no es necesario. El funcionamiento de StAX consiste en flujos de eventos, lo que, al igual que en SAX, evita la construcción explícita y el almacenamiento del modelo de objetos de un documento XML que son necesarios para los marcos orientados a árboles, realizando directamente operaciones asociadas a los eventos que van estando disponibles durante el análisis del documento XML. Este análisis puede realizarse mediante dos métodos: mediante *Cursor API* e *Iterator API*. El primero se comporta como un cursor que recorre el documento, de principio a fin, apuntando en todo momento únicamente a un elemento XML. El avance del cursor y obtención de datos del elemento XML al que apunta, se realiza a través de los métodos de la interfaz *XMLStreamReader*. El segundo representa un documento XML como un conjunto de objetos de eventos que son solicitados por la aplicación de procesamiento o procesador XML, los cuales representan tipos de elementos de información XML que implementan *XMLEvent* (como elemento abstracto de iteración) y que son accedidos y recorridos mediante los métodos de la interfaz *XMLEventReader*.

2.3.4 Enfoques específicos al procesamiento de documentos XML

Los enfoques específicos para el procesamiento de documentos XML se presentan como soluciones enfocadas a las tareas de procesamiento más usuales que se realizan sobre los documentos XML, como es la consulta de información en dichos documentos o la transformación de estos en función de un esquema respecto a otro. Los enfoques más destacables, que modelizan los documentos XML en estructuras arborescentes para adecuarlos a las tareas de procesamiento, son los siguientes:

- **Consulta XQuery:**

```
<NombresFormas>
  for $x in /Formas/Forma/Nombre
  return <Nombre>$x/text()</Nombre>
</NombresFormas>
```

- **Resultado:**

```
<NombresFormas>
  <Nombre>Cuadrado 10px</Nombre>
</NombresFormas>
```

Figura 2.3.5. Consulta XQuery y documento XML resultado que se obtiene de aplicar la misma al documento XML de ejemplo.

- XPath [Clark et al. 1999]. Es un lenguaje diseñado para operaciones de consulta de contenidos XML mediante la especificación de expresiones regulares aplicadas a rutas de acceso sobre los mismos [Abiteboul et al. 2000]. Las consultas se realizan mediante búsqueda y selección de nodos en los árboles que modelizan los documentos XML, empleando una sintaxis de rutas. Esta sintaxis se resume en emplear el símbolo "/" para formar el conjunto ordenado de elementos actual, que inicialmente quedaría formado por los nodos hijos de la raíz del árbol. Añadiendo un nombre de elemento a la expresión como sufijo, e indiciando entre "[" y "]" la expresión de condición de selección (que pueden incluir funciones), es posible seleccionar el nodo (o nodos) hijo indicado en dicho conjunto. Y así sucesivamente. Por ejemplo, mediante la expresión */Formas/Forma[tipo="regular"]*, aplicada al documento de la Figura 2.3.1, se obtienen aquellos elementos *<Forma>* de tipo *regular*. Adicionalmente, existen otros símbolos especiales como "@", que se utiliza para acceder a los atributos del elemento actual, "//" para realizar un salto de varios niveles de nodos, o "|" para unificar los resultados de varias expresiones.

- XQuery [Draper et al. 2003]. Es un lenguaje de consultas para documentos XML propuesto por el W3C mediante expresiones XPath y FLWOR: sentencias de tipo *For*, *Let*, *Where*, *Order-by* y *Return*. A través de las cinco cláusulas se consigue realizar operaciones de consulta y de transformación de documentos XML mediante los resultados obtenidos. Las cláusulas operan con expresiones XPath, aumentando las capacidades de éste de la siguiente manera: mediante *for* se seleccionan todos los elementos dados por una expresión XPath y se proporcionan en una variable, mediante *let* se asigna el resultado a una variable concreta, mediante *where* se seleccionan los elementos en función de una expresión, mediante *order* se define un criterio de ordenación de los elementos, y mediante *return* se construyen resultados XML. Por último, indicar que XQuery se caracteriza por poseer un sistema de tipado estático basado en XML Schema. La Figura 2.3.5 muestra un ejemplo de consulta XQuery y el documento resultado de aplicar dicha consulta al documento XML de la Figura 2.3.1.

- **Plantilla XSLT:**

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes" encoding="iso-8859-1"/>

  <xsl:template match="Formas">
    <html>
      <head><title>Formas</title></head>
      <body><xsl:apply-templates select="Forma"/></body>
    </html>
  </xsl:template>

  <xsl:template match="Forma">
     <h1><xsl:value-of select="Nombre"/></h1>
     <h2><xsl:value-of select="@tipo"/></h2>
     <xsl:apply-templates select="Puntos"/>
  </xsl:template>

  <xsl:template match="Puntos">
    <ol>
     <xsl:for-each select="Punto2D">
       <li>(<xsl:value-of select="CoordenadaX"/>,<xsl:value-of select="CoordenadaY"/>)</li>
     </xsl:for-each>
    </ol>
  </xsl:template>
</xsl:stylesheet>
```

- **Resultado:**

```
<html>
  <head><title>Formas</title></head>
  <body>
    <h1>Cuadrado 10px</h1>
    <h2>regular</h2>
    <ol>
      <li>(10,10)</li>
      <li>(20,10)</li>
      <li>(20,20)</li>
      <li>(10,20)</li>
    </ol>
  </body>
</html>
```

Figura 2.3.6. Transformación XSLT y documento XML resultado que se obtiene de aplicar la misma al documento XML de ejemplo.

- XSLT [Clark 1999, Kay 2007]. Es un lenguaje de transformación de documentos XML recomendado por el W3C, basado en *plantillas*. Estas plantillas utilizan expresiones XPath para seleccionar y operar sobre ciertos contenidos de los documentos XML, y permiten construir fragmentos XML mediante la aplicación de funciones sobre la representación estructural arborescente del documento, la extracción de contenido del documento original o la nueva adición de contenido, y otras operaciones más avanzadas. Los elementos de XSLT permiten la construcción de las plantillas mediante <xsl:template> y su atributo *match* permite asociar dicha plantilla a elementos XML mediante una expresión XPath. Mediante <xsl:value-of> se obtiene el valor textual de los elementos seleccionado en su atributo *select*. Así mismo, existen otras operaciones avanzadas como <xsl:for-each> para seleccionar un conjunto de elementos, o para la aplicación de las plantillas definidas a un elemento actual mediante <xsl:apply-templates>. El lenguaje no posee un sistema de tipos, aunque sí existen propuestas,

como la de aplicación de tipado estático descrita en [Tozawa 2001]. La Figura 2.3.6 muestra un ejemplo de transformación XSLT sobre el documento XML de la Figura 2.3.1.

2.4 Modelos lingüísticos al procesamiento de documentos XML

Anteriormente se han expuesto las técnicas más destacables que permiten describir, y en última instancia hacer efectivas, las tareas de procesamiento que se aplican a los documentos XML. No obstante, estas técnicas no aprovechan el enfoque lingüístico que se emplea en la construcción de traductores convencionales. Efectivamente, XML puede entenderse como un *metalenguaje* de marcado, que permite, a su vez, definir lenguajes de marcado sobre diferentes dominios mediante gramáticas documentales. Como consecuencia, surgen técnicas que se benefician de la potencia expresiva que brindan los formalismos básicos o avanzados de gramáticas para la generación de procesadores de lenguaje orientados a tipos de documentos XML. Las técnicas y herramientas existentes más representativas que emplean esquemas de traducción para la descripción de las tareas de procesamiento, se analizarán en la sección 2.4.1, mientras que aquellas que utilizan el formalismo avanzado de gramáticas de atributos se detallarán en la sección 2.4.2.

2.4.1 Especificación del procesamiento de documentos XML mediante esquemas de traducción

Los esquemas de traducción para especificar el procesamiento de documentos XML atienden a una manera de añadir instrucciones de procesamiento o fragmentos de código para que se ejecuten acciones en función de los distintos elementos del documento XML que son analizados. Para ello, este procesamiento se puede caracterizar con gramáticas y acciones semánticas que permitan ejecutar acciones a medida que se realiza la lectura de los elementos del documento XML. Como ejemplos más representativos caben destacar ANTXR y RelaxNGCC:

- ANTXR o ANother Tool for XML Recognition [Stanchfield 2005] es una especialización para XML del generador de traductores ANTLR (sección 2.2.4.1). Las especificaciones ANTXR son gramáticas incontextuales con fragmentos de código embebidos en las reglas de producción. Estos fragmentos de código se ejecutan a medida que los distintos elementos de dichas reglas son reconocidos. El lenguaje de programación que se emplea para realizar esta codificación es Java, obteniendo traductores en Java con soporte XML. La Figura 2.4.1 muestra un ejemplo de especificación ANTXR para el lenguaje XML de ejemplo, y para una tarea de procesamiento simple consistente en crear una lista con todas las figuras descritas Como puede observarse, el lenguaje ANTXR deriva de ANTLR, y es muy similar al de JavaCC. Sin embargo, respecto a éste último, se fuerza a utilizar las etiquetas XML como símbolos no terminales de la gramática, es decir, como símbolos cabeza de producciones y símbolos del cuerpo de los mismos. Esto supone un ligamento implícito de la estructura de la especificación

con la estructura del propio documento XML, lo que puede convertirse en un problema a la hora de querer utilizar una gramática con diferente estructura.

```
header {
  import Forma;
  import Punto;
  import java.util.ArrayList;
}

public class Traductor extends Parser;

document returns [ArrayList<Forma> result = null]
  : result = <Formas> EOF
  ;
<Formas> returns [ArrayList<Forma> formas = new ArrayList<Forma>()] {Forma forma;}
  : (forma = <Forma> {formas.add(forma);} )*
  ;
<Forma> returns [Forma forma = null] {ArrayList<Punto> puntos = new ArrayList<Punto>();}
  : n = <Nombre> <Puntos>[puntos] {forma = new Forma(n, @tipo, puntos);}
  ;
<Nombre> returns [String v = null]
  : pcdata:PCDATA {v = pcdata.getText();}
  ;
<Puntos> [ArrayList<Punto> puntos] {Punto p;}
  : (p = <Punto> {puntos.add(p);} )*
  ;
<Punto2D> returns [Punto p = null] {String x, y;}
  : x = <CoordenadaX> y = <CoordenadaY> {p = new Punto(x, y);}
  ;
<CoordenadaX> returns [String v = null]
  : pcdata:PCDATA {v = pcdata.getText();}
  ;
<CoordenadaY> returns [String v = null]
  : pcdata:PCDATA {v = pcdata.getText();}
  ;
```

Figura 2.4.1. Especificación ANTXR para el documento XML de ejemplo.

- RelaxNGCC o REgular LAnguage for XML Next Generation Compiler Compiler [Okajima 2002] es un generador de traductores a partir de especificaciones gramaticales basado en RELAX NG (sección 2.3.2). Se rige por el mismo principio de añadir semántica en las especificaciones gramaticales a base de introducir fragmentos de código en Java en las mismas. Las especificaciones RelaxNGCC son una especialización del lenguaje de esquema XML RELAX NG, en donde las acciones semánticas son fragmentos de código que se embeben en los elementos del esquema, para ser ejecutados una vez son reconocidos. De esta manera, las tareas de procesamiento descritas mediante fragmentos de código embebido serán realizadas conforme se vayan reconociendo los distintos elementos XML de un documento. Bajo este principio, RelaxNGCC genera los traductores en Java con soporte a documentos XML utilizando un marco de procesamiento orientado a eventos (SAX, sección 2.3.3.2). La Figura 2.4.2 muestra un ejemplo de especificación RelaxNGCC para el documento XML de ejemplo. Por otro lado, la alternativa RelaxNGCC, si bien se muestra como una solución de generación de traductores mediante un enfoque dirigido por la sintaxis del lenguaje, presenta como desventaja que la descripción del procesamiento queda acoplada directamente a la

definición documental y, por lo tanto, a dicha estructura, como sucede en el ejemplo de la anterior figura.

```
<?xml versión="1.0" encoding= "utf-8">
<grammar xmlns="http://relaxng.org/ns/structure/1.0" xmlns:c="http://www.xml.gr.jp/xmlns/relaxngcc">
  <start c:class="Formas">
    <c:java-import>
    import Forma
    import Punto;
    import java.util.ArrayList;
    </c:java-import>

    <c:java-body>
    ArrayList<Forma> formas = new ArrayList<Forma>();
    </c:java-body>

    <element name="Formas">
      <zeroOrMore>
        <element name="Forma">
          <attribute name="tipo"><data type=”string” c:alias="t"/></attribute>
          <element name="Nombre"><text c:alias="n"/></element>
          <c:java>ArrayList<Punto> puntos = new ArrayList<Punto>();</c:java>
          <element name=”Puntos”>
            <zeroOrMore>
              <element name="Punto2D">
                <element name="CoordenadaX"><text c:alias="x"/></element>
                <element name="CoordenadaY"><text c:alias="y"/></element>
                <c:java>puntos.add(new Punto(x, y));</c:java>
              </element>
            </zeroOrMore>
          </element>
          <c:java>formas.add(new Forma(n,t, puntos));</c:java>
        </element>
      </zeroOrMore>
    </element>
  </start>
</grammar >
```

Figura 2.4.2. Especificación RelaxNGCC para el documento XML de ejemplo.

Ambas propuestas presentan, por tanto, la desventaja principal de no poder aportar una gramática que permita describir el procesamiento con una estructura desacoplada e independiente a la estructura del documento XML, y que, a su vez, sea conforme a éste.

2.4.2 Especificación del procesamiento de documentos XML mediante gramáticas de atributos

El lenguaje descriptivo empleado en XML posee una naturaleza similar al formalismo de gramáticas incontextuales. Se podría identificar los distintos elementos XML de la siguiente manera: etiquetas de apertura y cierre de un elemento como símbolos no terminales cabeza de producción, los elementos internos o hijos directos que se definen en él como no terminales de los cuerpos de las producciones, los atributos asociados a estos elementos como semántica operacional, y los otros elementos con atributos de lectura (por ejemplo #PCDATA) como terminales con su información dada por atributos léxicos. Visto de esta manera, se induce la posibilidad de emplear gramáticas de atributos para poder describir el procesamiento de los documentos XML. No obstante, la aplicación directa no es posible. Por

una parte, la semántica operacional empleada en las gramáticas de atributos no puede ser descrita directamente en XML al no poderse especificar la síntesis y herencia de atributos ni su procesamiento. Por otra parte, la notación empleada en las gramáticas de atributos es de tipo BNF mientras que los documentos XML se rigen por especificaciones extendidas o EBNF en sus descripciones documentales asociadas (p.e., DTDs). Es por ello que han aparecido distintas propuestas al procesamiento de documentos XML mediante gramáticas de atributos que abordan estos problemas. En las siguientes secciones se detallarán las propuestas más destacables, en función del enfoque que adoptan: asociar la semántica de atributos y ecuaciones externamente (sección 2.4.2.1), transformar la gramática documental al formalismo BNF usado en las gramáticas de atributos (sección 2.4.2.2), extender el modelo básico de gramáticas de atributos al modelo EBNF (sección 2.4.2.3) y, adaptar las estructuras arborescentes de los documentos XML a las necesidades de las gramáticas de atributos (sección 2.4.2.4).

2.4.2.1 Desacoplamiento de semántica

Esta propuesta consiste en dotar a los documentos XML de semántica de procesamiento a través de un documento externo (por ejemplo, otro documento XML), mediante reglas semánticas y asociación de dichas reglas a los correspondientes elementos del documento. De esta manera, la semántica queda desacoplada de la gramática documental XML, sin alterar su contenido original. No obstante, al estar orientada la asociación de reglas semánticas a elementos descontextualizados o, a lo sumo, a ciertos patrones sencillos sobre los árboles documentales, el enfoque no permite aprovechar la potencia estructural para la descripción del procesamiento que una gramática de atributos sí aporta, al quedar ésta definida sobre la estructura sintáctica del lenguaje. Como ejemplos de esta propuesta destacan los siguientes metalenguajes XML para la descripción del procesamiento de documentos XML mediante la asociación de atributos y ecuaciones semánticas con tipos de elementos XML:

- SRD o Semantic Rule Definition [Psaila & Crespi-Reghizzi 1999]. Mediante un tipo de atributos denominados *atributos intensionales*, distinguidos de los atributos documentales denominados *atributos extensionales*, los atributos semánticos se asocian a los elementos XML. Mediante reglas semánticas se establece la forma de cómputo y el valor de dichos atributos, cuyo tipo se extiende a un conjunto más amplio que el soportado por la definición documental. Concretamente, el metalenguaje presenta tres partes: (i) permite definir nuevos tipos para los atributos intensionales mediante las etiquetas <recordtype> (tipo complejo) y <settype> (lista/conjunto de tipo especificado) encapsuladas en <types> de manera opcional, (ii) permite asociar dichos atributos a los distintos elementos XML del lenguaje mediante <add-to-element> y <attribute> encapsulados en <intensional-attributes>, y (iii) permite asociar a dichos atributos las reglas semánticas que lo computan mediante una estructura de nodos *padre-hijo* encapsulada en <semantic-rules>. Esta última parte se formula mediante elementos <rules-for> para definir un conjunto de reglas semánticas, que se asocian a un elemento determinado a través de sus atributos. Tales reglas pueden declararse

```
<?xml versión=”1.0”?> <!DOCTYPE SRD SYSTEM “srd.dtd”>
<types>
 <recordtype name=”forma”>
  <field name=”nombre” type=”string”/> <field name=”tipoForma” type=”string”/> <field name=”puntos” type=”set”/>
 </recordtype> <settype name=”formas” type=”forma”>
 <recordtype name=”punto”>
  <field name=”x” type=”integer”/> <field name=”y” type=”integer”/>
 </recordtype> <settype name=”puntos” type=”punto”>
</types>

<intensional-attributes>
 <add-to-element name=”Formas”> <attribute name=”formas” type=”formas”/> </add-to-element>
 <add-to-element name=”Forma”>
  <attribute name=”formas_h” type=”formas”/> <attribute name=”formas” type=”formas”/>
  <attribute name=”nombre” type=”string”/> <attribute name=”forma” type=”string”/>
 </add-to-element>
 <add-to-element name=”Puntos”> <attribute name=”puntos” type=”puntos”/> </add-to-element>
 <add-to-element name=”Nombre”> <attribute name=”nombre” type=”string”/> </add-to-element>
 <add-to-element name=”Punto2D”>
  <attribute name=”puntos_h” type=”puntos”/> <attribute name=”puntos” type=”puntos”/>
  <attribute name=”coord” type=”integer”/> <attribute name=”punto” type=”punto”/>
 </add-to-element>
 <add-to-element name=”CoordenadaX”> <attribute name=”x” type=”integer”/> </add-to-element>
 <add-to-element name=”CoordenadaY”> <attribute name=”y” type=”integer”/> </add-to-element>
</intensional-attributes>

<semantic-rules>
 <rules-for element=”Formas” father-of=”Forma”> <conditional> <if> <isFirst element=”Forma”/>
      <derive attribute=”Forma.formas_h” from=”#EMPTYSET(forma)”/>
      <derive attribute=”Formas.formas” from=”Forma.formas”/> </if> <else/>
       <derive attribute=”Forma.formas_h” from=”#PRED(Forma).formas”/>
       <derive attribute=”Formas.formas” from=”Forma.formas”/>
 </conditional> </rules-for>
 <rules-for element=”Forma” father-of=”Nombre”> <static>
  <derive attribute=”Forma.nombre” from=”#CALL.Text(#PCDATA.Content)”/>
 </static> </rules-for>
 <rules-for element=”Forma” father-of=”Puntos”> <static>
  <derive attribute=”Forma.forma” from=”#RECORD(Forma.nombre, Forma.tipo, Puntos.puntos)”/>
  <derive attribute=”Forma.formas” from=”#ADDTOSET(Forma.formas_h, Formas.forma)”/>
 </static> </rules-for>
 <rules-for element=”Puntos” father-of=”Punto2D”> <conditional> <if> <isFirst element=” Punto2D”/>
      <derive attribute=”Punto2D.puntos_h” from=”#EMPTYSET(puntos)”/>
      <derive attribute=”Puntos.puntos” from=” Punto2D.puntos”/> </if> <else/>
       <derive attribute=”Punto2D.puntos_h” from=”#PRED(punto).puntos”/>
       <derive attribute=”Puntos.puntos” from=” Punto2D.puntos”/>
 </conditional> </rules-for>
 <rules-for element=”Punto2D” father-of=”CoordenadaX”> <static>
  <derive attribute=”Punto2D.x” from=”CoordenadaX.x”/>
 </static> </rules-for>
 <rules-for element=”Punto2D” father-of=”CoordenadaY”> <static>
  <derive attribute=”Punto2D.punto” from=”#RECORD(Punto.x, CoordenadaY.y)”/>
  <derive attribute=”Punto2D.puntos” from=”#ADDTOSET(Punto2D.puntos_h, Punto2D.punto)”/>
 </static> </rules-for>
 <rules-for element=”Nombre” father-of=”#PCDATA”> <static>
  <derive attribute=”Nombre.nombre” from=”#CALL.Text(#PCDATA.Content)”/>
 </static> </rules-for>
 <rules-for element=”CoordenadaX” father-of=”#PCDATA”> <static>
  <derive attribute=”CoordenadaX.x” from=”#CALL.enNum(#PCDATA.Content)”/>
 </static> </rules-for>
 <rules-for element=”CoordenadaY” father-of=” PCDATA”> <static>
  <derive attribute=”CoordenadaY.y” from=”#CALL.enNum(#PCDATA.Content)”/>
 </static> </rules-for>
</semantic-rules>
```

Figura 2.4.3. Documento SRD para el documento XML de ejemplo.

como estáticas mediante <static>, o declararse como condicionales si su aplicación depende de una forma determinada en la que se presente la estructura de los hijos del elemento, lo que permite aplicar una u otra regla en función del contenido de los mismos. La declaración de la regla semántica en sí se realiza mediante la etiqueta <derive>, cuyo atributo semántico se define mediante el atributo de la etiqueta *attribute*, y su computación se describe refiriendo valores de otros atributos intensionales, extensionales, y funciones externas (#CALL), empleando para ello un sencillo lenguaje de expresiones en su atributo *form*. La Figura 2.4.3 muestra un ejemplo.

```
<semantic-rules>
 <rules-for root="Formas">
  <rule element=”srml:root" attrib="formas”>
     <expr><extern-function name="creaFormas”/></expr>
  </rule>
 <rule element="Forma" attrib=”formash">
     <expr><attribute element="srml:root" attrib="formas"/></expr>
 </rule>
</rules-for>
<rules-for root=”Forma“>
  <rule element="srml:root" attrib="formas">
   <expr>
    <extern-function name="añadeForma">
     <param><expr><attribute element="srml:root" attrib="formash"/></expr></param>
     <param><position/></param>
     <param> <expr> <value-ref path="Nombre"/> </expr> </param>
     <param> <expr> <value-ref path="@tipo"/> </expr> </param>
     <param> <expr> <attribute element="Puntos"  attrib="puntos"> </expr></param>
   </expr>
  </rule>
 </rules-for>
 <rules-for root="Puntos">
  <rule element=”srml:root” attrib="puntos">
     <expr><extern-function name="creaPuntos”/></expr>
  </rule>
  <rule element=”Punto2D” attrib="puntosh">
     <expr><attribute element="srml:root" attrib=”puntosh"/></expr>
  </rule>
 </rules-for>
 <rules-for root="Punto2D">
  <rule element="srml:root" attrib="puntos">
   <expr>
    <extern-function name="añadePunto">
     <param><expr><attribute element="srml:root" attrib="puntosh"/></expr></param>
     <param><position/></param>
     <param> <expr> <value-ref path="CoordenadaX"/> </expr> </param>
     <param> <expr> <value-ref path="CoordenadaY"/> </expr></param>
   </expr>
  </rule>
 </rules-for>
</semantic-rules>
```

Figura 2.4.4. Documento SRML para el documento XML de ejemplo.

- SRML o Semantic Rule MetaLanguage [Havasi 2002]. Una alternativa a la propuesta SRD, siguiendo el mismo concepto, pero más simple, pues no permite declarar tipos compuestos para los atributos semánticos y estos se asocian directamente en la declaración de las reglas semánticas. De esta manera, las descripciones se reducen directamente a la parte descriptiva de las reglas semánticas encapsuladas en

<semantic-rules>. Las diferencias respecto a SRD son las siguientes: (i) en SRML las reglas semánticas se asocian a elementos XML y no a nodos *padre-hijo*, y las ecuaciones semánticas se describen mediante elementos XML y no mediante un sencillo lenguaje de expresiones embebido; (ii) las reglas semánticas se definen para un elemento XML encapsuladas en <rules-for> mediante <rule> para describir el atributo semántico, y mediante <expr> para describir su expresión semántica de cómputo (estas expresiones semánticas pueden referir a otros atributos o a funciones externas), y (iii) los atributos se refieren con <attribute>, y las funciones externas se describen mediante <extern-function>, en donde cada uno de sus parámetros (son expresiones semánticas a su vez) se describen mediante <param>. La Figura 2.4.4 muestra un ejemplo.

2.4.2.2 Transformación de gramáticas

El formalismo básico de las gramáticas de atributos utiliza BNF. Sin embargo, las gramáticas documentales utilizan EBNF. Por lo tanto, las gramáticas documentales XML no son directamente aptas para extenderlas con semántica y obtener así una gramática de atributos para describir el procesamiento. Como solución a este problema, es posible aplicar alguna técnica para transformar la gramática documental EBNF a una gramática BNF equivalente apta para incluir en ella la semántica que caracteriza el procesamiento. La técnica propuesta por [Gançarski et al. 2002, Gançarski et al. 2006] es un ejemplo representativo, en donde los diferentes elementos de una gramática documental se analizan para convertir sus reglas EBNF en BNF mediante la aplicación de reglas predefinidas de transformación, de la siguiente manera:

- Los elementos formados por una secuencia de *n* elementos en su modelo de contenidos, como <!ELEMENT elemType (e_1, …, e_n)> se transforman en la producción elemType ::= e_1 … e_n, y para los formados por una secuencia de *n* elementos alternados como <!ELEMENT elemType (e_1 | … |e_n)>, se crea una producción para cada elemento elemType ::= e_1, …, elemType ::= e_n.

- Si el modelo de contenidos de un elemento presenta una forma más compleja, se crea la producción elemType ::= NT, donde NT es un nuevo símbolo no terminal NT cuya gramática subyacente representa la expresión regular del modelo de contenidos. Para la transformación de dicha expresión regular se aplican reglas de transformación equivalentes de casos básicos y compuestos. Por ejemplo, para transformar (exp*) se crean las producciones NT_{exp*} ::= NT_{exp} NT_{exp*}, NT_{exp*} ::= λ, y para transformar (exp+) las producciones NT_{exp+} ::= NT_{exp} NT_{exp+}, NT_{exp+} ::= NT_{exp}.

La Figura 2.4.5 muestra el resultado de aplicar la transformación a la DTD del lenguaje de ejemplo. Como puede observarse, el resultado obtenido es una gramática BNF con una estructura heredada de las reglas de transformación, que podrá acondicionarse con atributos y ecuaciones semánticas para convertirse en una gramática de atributos. Sin embargo, el

resultado no es necesariamente aquel que brinda la estructura deseada y óptima para incluir la semántica de cada tarea de procesamiento particular a realizar.

```
Formas ::=  LFormas
LFormas ::= Forma LFormas | λ
Forma ::= Nombre Puntos
Puntos ::= Punto2D LPuntos Punto2D
LPuntos ::= Punto2D LPuntos | Punto2D
Punto2D ::= CoordenadaX CoordenadaY
CoordenadaX ::= text
CoordenadaY ::= text
```

Figura 2.4.5. Gramática BNF resultante de transformar la DTD de ejemplo.

2.4.2.3 Extensiones del modelo básico

Como contrapartida a la transformación de gramáticas EBNF en BNF, surge la propuesta inversa: extender las gramáticas de atributos para dar soporte a EBNF. De entre estas propuestas, son representativas las gramáticas de atributos extendidas, y las gramáticas de atributos orientadas a flujos XML. A continuación, se revisan dichas propuestas.

2.4.2.3.1 Gramáticas de atributos extendidas

Las gramáticas de atributos extendidas o EAGs [Neven 1999, Neven 2005] son gramáticas de atributos con notación EBNF en el cuerpo de sus producciones. De esta forma, las ecuaciones semánticas se adaptan para el manejo de EBNF con tres componentes: un *selector de valores de atributos* en función del contexto, un *predicado de aplicabilidad* para evaluar si la ecuación es aplicable en el contexto, y una *función semántica* para el cómputo y obtención del valor de la ecuación semántica. Como resultado, se puede describir un procesamiento compatible a la definición documental XML empleando las gramáticas de atributos.

La Figura 2.4.6 muestra un ejemplo. En dicho ejemplo:

- La selección de valores de atributos se lleva a cabo indicando, entre <…> los atributos a considerar. De esta forma, para aplicar la ecuación se recorren los hijos del nodo en orden, se recolectan los valores de los atributos para cada símbolo referido en listas, y dichas listas se concatenan en el orden en el que aparecen en el selector. Así, por ejemplo, el selector <Puntos2D$_0$, Puntos2D$_1$, Puntos2D$_2$> en el contexto de la producción Puntos ::= Punto2D Punto2D+ Punto2D formaría, como valor sobre el que operar en la ecuación, una lista formada por el primer punto, la secuencia del resto de puntos menos el último, y el último punto.

- El predicado de aplicabilidad se representa mediante una función que actúa sobre el valor seleccionado, y decide si la ecuación es o no aplicable (se utiliza la notación *lambda* para describir dicha función). De esta forma, es posible tener varias ecuaciones para un mismo atributo, válidas en distintos contextos. En este caso, todos los

predicados de aplicabilidad evalúan a *cierto*, ya que existe una única ecuación para cada atributo.

- Por último, la función semántica se representa como una función (utilizando, de nuevo, la notación *lambda*), que, tomando como entrada la lista de valores seleccionada, lleva a cabo el cómputo. Así, por ejemplo, la función semántica λv.nuevaForma(v_0, Forma@tipo, $v_{[1..]}$) indica la construcción de una nueva forma (aplicando la función *nuevaForma*, que no se detalla aquí), sobre: (i) el nombre de la forma (como queda patente en la gramática de la Figura 2.4.6, vendrá dada por el primer componente de la lista de valores seleccionados v_0), (ii) el tipo de la forma (valor del atributo *tipo* de la etiqueta <Forma>, que se consulta aquí como Forma@tipo), y (iii) la lista de puntos (el resto de valores, denotado como $v_{[1..]}$).

Formas ::= Forma*
  Formas.formas = [<Forma.forma>, λv.**true** , λv.v]
Forma ::= Nombre Puntos
  Forma.forma = [<Nombre.nombre, Puntos.lpuntos>, λv.**true**, λv.nuevaForma(v_0, Forma@tipo, $v_{[1..]}$)]
Nombre ::= **text**
  Nombre.nombre = [<**text**.text>, λv.**true**, λv.v]
Puntos ::= Punto2D Punto2D+ Punto2D
  Puntos.lpuntos = [<$Punto2D_0$.punto, $Punto2D_1$.punto, $Punto2D_2$.punto>, λv.**true**, λv.v]
Punto2D ::= CoordenadaX CoordenadaY
  Punto2D.punto = [<CoordenadaX.val, CoordenadaY.val>, λv.**true**, λv.nuevoPunto(v_0, v_1)]
CoordenadaX ::= **text**
  CoordenadaX.val = [<**text**.text>, λv.**true**, λv.enNum(v)]
CoordenadaY ::= **text**
  CoordenadaY.val = [<**text**.text>, λv.**true**, λv.enNum(v)]

Figura 2.4.6. Gramática EAG para el documento XML de ejemplo.

Para finalizar, nótese que las expresiones EBNF de las producciones deben ser no ambiguas para evitar la mezcla de contextos [Brüggemann-Klein et al. 1998]. Nótese, así mismo, que, aunque el enfoque se formuló para la realización eficiente de consultas en los documentos XML, permite también su aplicación a la realización de tareas de procesamiento ajenas a ese dominio.

2.4.2.3.2 Gramáticas de atributos orientadas a flujos XML

Las gramáticas de atributos orientadas a flujos XML o XSAGs [Koch & Scherzinger 2007] constituyen una propuesta de extensión de gramáticas de atributos para aquellos documentos XML que son generados incrementalmente, como flujos de datos XML. Son más eficientes en este sentido, pero se restringen a una subclase de gramáticas de atributos con restricciones en los atributos que pueden poseer los distintos elementos: *gramáticas L-atribuidas* [Wilhelm 1982]. Estas gramáticas se formulan como una extensión del modelo básico para soportar la notación EBNF. La propuesta de XSAGs propone dos mecanismos de especificación del procesamiento XML: Basic XSAG o bXSAG donde exclusivamente los elementos poseerán atributos y, easY XSAG o yXSAG donde también las expresiones de los modelos de contenidos XML pueden poseer atributos. De esta forma, respecto a bXSAG, yXSAG aporta un

enriquecimiento estructural del árbol de análisis sintáctico albergando la subestructura que se deriva de las expresiones regulares de los modelos de contenidos.

```
Formas ::= [formas = []]
  (
  Forma
    [formas = añadeA(formas, forma)]
  )*
Forma ::= [tipo = @tipo]
  Nombre Puntos
    [forma = nuevaForma(nombre, tipo, puntos)]
Nombre ::=
  text
    [nombre = text]
Puntos ::=
  Punto2D
    [puntos = [punto]]
  (
  Punto2D
    [puntos = añadeA(puntos, punto)]
  )+
  Punto2D
    [puntos = añadeA(puntos, punto)]
Punto2D ::=
  CoordenadaX CoordenadaY
    [punto = nuevoPunto(x, y)]]
CoordenadaX ::=
  text
    [x = enNum(text)]
CoordenadaY ::=
  text
    [y = enNum(text)]
```

Figura 2.4.7. Gramática yXSAG para el documento XML de ejemplo.

El modelo de evaluación semántica utilizado en esta propuesta se entiende como una propagación de un único atributo heredado *h*, y la recolección de un único atributo sintetizado *s*, para cada nodo del árbol de análisis sintáctico. Esta propagación se realiza de una manera secuencial entre nodos, en donde el atributo sintetizado de un nodo padre se obtiene de su último nodo hijo, y su atributo heredado se inyecta al primer nodo hijo. Entre nodos hijos hermanos, los atributos heredados del siguiente hermano se corresponden a los atributos sintetizados del nodo hermano actual, y así sucesivamente. Mediante este mecanismo, es posible liberar espacio de los atributos que ya no son necesarios en base a cómo se establezcan y propaguen los valores en las tuplas. Los valores de los atributos se definen a través de tuplas de una dimensión prestablecida y se establecen mediante un *mecanismo de atribución*. Por defecto se aplica la atribución identidad, que no alterna el valor del flujo de atributos, pero este comportamiento puede modificarse especificando un mecanismo de atribución diferente explícitamente en puntos seleccionados de la especificación.

La Figura 2.4.7 muestra un ejemplo de yXSAG para el lenguaje ejemplo y procesamiento considerados. Como componentes de las tuplas propagadas mediante los atributos *h* y *s* se consideran: (i) la lista de formas (se referirá mediante *formas*), (ii) una forma concreta (*forma*), (iii) un nombre (*nombre*), (iv) una lista de puntos (*puntos*), (v) un punto (*punto*), (vi) el tipo de la forma (*tipo*), (vii) la coordenada x (*x*), y (viii) la coordenada y (*y*). En aquellos puntos en los que se precisa un cambio en el mecanismo de atribución por defecto, se muestra cómo

actualizar dichos componentes en función de los actuales. Así, por ejemplo, en la producción Formas ::= Forma*:

- Antes de entrar en la lista de elementos <Forma>, es necesario fijar la lista de formas a la lista vacía (formas = [])
- Cada vez que se termina de visitar una forma, es necesario añadir la forma resultante a la lista: formas = añadeA(formas, forma). En este caso, *formas* en la parte izquierda será el componente actualizado en la tupla del correspondiente atributo sintetizado, mientras que *formas* y *forma* en la parte derecha serán los componentes recibidos en el correspondiente atributo heredado.

De esta manera, como el ejemplo evidencia, este tipo de especificación obliga a pensar en el flujo preestablecido de atributos en el árbol de análisis, y, por tanto, en un orden particular de evaluación de atributos, lo que supone un cierto demérito a la capacidad de abstracción del formalismo de las gramáticas de atributos, en favor de mayor eficiencia en el procesamiento de flujos XML.

2.4.2.4 Transformación de árboles documentales

Por último, existen propuestas que se centran en la adaptación de las estructuras arborescentes producidas por los documentos XML a las necesidades de las gramáticas de atributos. Concretamente para flujos XML, en [Nishimura & Nakano 2005] se propone la transformación de flujos XML mediante varias gramáticas de atributos para transformar el flujo XML a una representación arborescente adecuada. Mediante una primera gramática de atributos genérica se define la transformación a una representación canónica de árboles del flujo XML. Mediante una segunda gramática de atributos se define la transformación particular que se desea aplicar. Por último, mediante una tercera gramática genérica se define la transformación al flujo XML de la representación canónica.

Así, por ejemplo, en el enfoque de [Nishimura & Nakano 2005, Nakano 2004] se utiliza una representación binaria de los árboles XML como se ilustra en la Figura 2.4.8. Como puede observarse, los nodos de esta representación son de dos tipos: nodos contenido que alojan contenido textual XML y, nodos elemento que representan elementos XML, y que tienen como máximo dos nodos hijos. Los nodos de este último tipo pueden apuntar al contenido del elemento en sí como primer nodo hijo, y como segundo nodo hijo a su nodo hermano si este existe. De esta manera, esta representación sí puede modelizarse mediante una gramática BNF, como la diseñada en la Figura 2.4.9. No obstante, dicha gramática puede no ser la más natural a la hora de especificar la semántica.

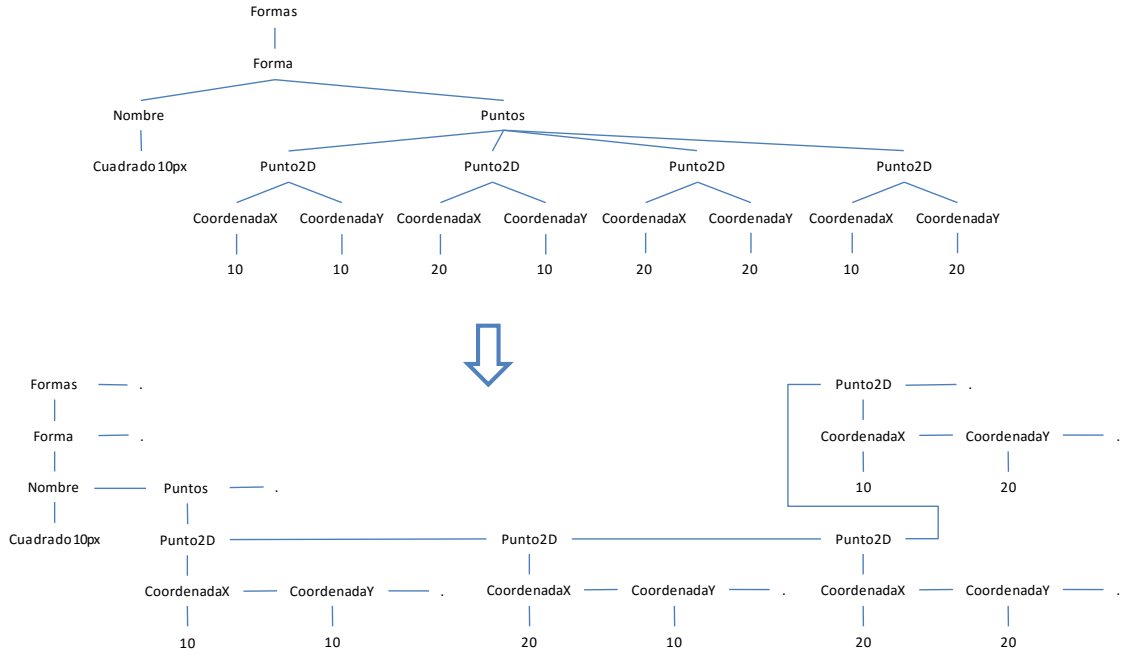


Figura 2.4.8. Transformación a árboles binarios del documento XML de ejemplo.

De esta forma, aunque el método propuesto puede ser automatizado e incluso evitar la construcción de los arboles explícitamente [Ganzinger & Giegerich 1984], la representación limita el tipo de formas estructurales a la de los árboles generados por la primera transformación de cara a la asociación de la semántica, lo que puede llevar a la elaboración de gramáticas poco naturales.

```
N0 ::= $Formas N1
N1 ::= $Forma N2 N1 | $Forma N2
N2 ::= $Nombre text N3
N3 ::= $Puntos N4
N4 ::= $Punto2D N5 N4| $Punto2D N5
N5 ::= $CoordenadaX text N6
N6 ::= $CoordenadaY text
```

Figura 2.4.9. Gramática BNF para el documento XML de ejemplo.

2.5 A modo de conclusión

Este capítulo ha enfatizado las interconexiones existentes entre dos enfoques, en principio, dispares: la construcción de procesadores de lenguajes, por una parte, y, por otra, el procesamiento de documentos XML.

En lo que se refiere al desarrollo de procesadores de lenguaje, el capítulo ha hecho patente como esta es una actividad muy madura, que puede beneficiarse del uso de formalismos de especificación de alto nivel y de meta-herramientas que soportan dichos formalismos para generar de forma efectiva implementaciones finales de los procesadores a partir de sus especificaciones. A este respecto, el uso de formalismos basados en esquemas de traducción,

tanto orientados a la especificación de traductores descendentes como orientados a la especificación de traductores ascendentes, junto con las correspondientes herramientas de soporte, se ha convertido en una práctica común en el diseño e implementación de lenguajes informáticos. Así mismo, el formalismo de las gramáticas de atributos, si bien no tan extendido, supone una alternativa declarativa de más alto nivel, y es, así mismo, susceptible de ser soportado por herramientas capaces de generar automáticamente implementaciones eficientes. El formalismo es, así mismo, especialmente apropiado para acomodar técnicas de modularización, ya que el modelo de evaluación semántica subyacente al formalismo es el encargado de reorganizar automáticamente el proceso de evaluación conforme se añaden nuevas características a las especificaciones.

Por otra parte, el capítulo ha puesto también de manifiesto cómo las tecnologías de procesamiento de documentos XML más habituales son tecnologías orientadas a datos. Efectivamente, dichas tecnologías entienden los lenguajes de marcado XML como estructuras de datos, los documentos como instancias de dichas estructuras, y los procesamientos como programas que operan sobre las mismas utilizando técnicas habituales de manipulación (recorridos, consultas, actualizaciones, etc.). A este respecto, el capítulo también ha diferenciado claramente entre *tecnologías específicas* (orientadas a realizar un tipo particular de procesamiento), y *tecnologías genéricas* (orientadas a realizar un rango de tareas mucho más general). Entre ambos polos existe, por tanto, la tensión usual entre la adopción de un enfoque específico (más fácil de usar, pero con un dominio de aplicabilidad mucho más reducido), y un enfoque genérico (aplicable en dominios muchos más amplios y no establecidos a priori, pero mucho más difícil de aplicar).

Como contraposición a los enfoques orientados a datos, en este capítulo se ha puesto de manifiesto también cómo existen enfoques que explotan la naturaleza lingüística de los lenguajes de marcado. Tales enfoques entienden, por tanto, las aplicaciones de procesamiento de documentos XML como tipos particulares de procesadores de lenguaje, proporcionando, de esta forma, el punto de interconexión entre los dos aspectos anteriores (desarrollo de procesadores de lenguaje y tecnologías de procesamiento de documentos XML). Como consecuencia, será posible utilizar formalismos específicos análogos a los utilizados en el campo del desarrollo de procesadores de lenguaje para construir aplicaciones de procesamiento de documentos XML. En esta línea, se han descrito propuestas basadas en esquemas de traducción, y también propuestas basadas en gramáticas de atributos. No obstante, la práctica totalidad de estos enfoques presentan un fuerte acoplamiento con las gramáticas documentales. De esta forma, las gramáticas documentales se toman como punto de partida para intercalar acciones semánticas en las propuestas basadas en esquemas de traducción, así como para añadir atributos y ecuaciones semánticas en las basadas en gramáticas de atributos (propuestas que, por otra parte, se han pensado para resolver tipos de tareas muy específicos, normalmente relativas a consultas).

Concebido el uso de las técnicas de desarrollo de procesadores de lenguaje como un enfoque genérico al desarrollo de aplicaciones de procesamiento XML, el acoplamiento con la gramática documental no tiene por qué ser necesariamente una característica deseable (no más que, por ejemplo, el acoplamiento de la implementación de un compilador para un

lenguaje de programación con la gramática EBNF o los diagramas sintácticos que describen su sintaxis en el correspondiente manual de usuario). Es por ello que en los trabajos que hemos llevado a cabo en el contexto del grupo ILSA, y, por tanto, en el trabajo llevado a cabo en esta tesis, hemos promovido el uso de gramáticas incontextuales independientes de las gramáticas documentales, con el fin de caracterizar estructuras sintácticas adaptadas a los requisitos particulares de las distintas tareas de procesamiento. Este aspecto se desarrolla con detalle en el próximo capítulo.

Capítulo 3
Desarrollo Dirigido por Lenguajes de Aplicaciones de Procesamiento XML

3.1 Introducción

En este capítulo se analizan los trabajos previos realizados en el Grupo de Investigación en Ingeniería de Lenguajes Software y Aplicaciones (ILSA) de la UCM en relación con el desarrollo dirigido por lenguajes de aplicaciones de procesamiento de documentos XML. Dichos trabajos propugnan el uso de métodos, técnicas y herramientas de desarrollo de procesadores de lenguaje para la construcción de este tipo de aplicaciones. Como caso de estudio de ejemplo, se utilizará el presentado en la sección 3.2. La sección 3.3 plantea cómo es posible realizar la construcción de aplicaciones de procesamiento XML complejas mediante la combinación de generadores de traductores con marcos convencionales de procesamiento XML siguiendo un enfoque de desarrollo óptimo. Posteriormente, en la sección 3.4 se presenta un entorno generador de aplicaciones de procesamiento XML, XLOP 1.0, y se demuestra que, mediante técnicas más avanzadas de procesamiento de lenguajes, se consigue realizar las tareas de procesamiento descritas mediante una gramática de atributos de una manera sencilla y eficiente, lo que convierte, a su vez, en un proceso sencillo el mantener y evolucionar las diversas posibles aplicaciones generadas mediante esta solución y con la ayuda de dicho entorno. Por último, en la sección 3.5 se discuten estas soluciones y se sientan las bases para una nueva propuesta basada en un nuevo modelo de gramáticas de atributos propuesto en esta tesis: las *gramáticas de atributos multivista*.

Los contenidos de este capítulo están basados en los siguientes trabajos: [Sarasa et al. 2012, Sarasa et al. 2009-a, Sarasa et al. 2009-b, Sarasa et al. 2011, Temprado et al. 2010-b].

3.2 Caso de estudio: <e-Tutor>

Como caso de estudio para ilustrar el proceso de desarrollo que se plantea en este capítulo, utilizaremos el sistema <e-Tutor>. <e-Tutor> es una aplicación de generación de tutores socráticos a partir de documentos XML [Sierra et al. 2008, Temprado et al. 2010-b]. Los usuarios o estudiantes de cierta materia que trabajan con un tutorial de <e-Tutor> deben resolver problemas siguiendo un sistema de diálogo discípulo-maestro. El sistema analiza las respuestas del estudiante y, en función de dichas respuestas, le aporta una realimentación constructiva y, decide cual será el siguiente problema a resolver. La realimentación depende de un estado de interacción, representado en <e-Tutor> como un conjunto de contadores: existe un contador para cada respuesta, el cual se incrementa cada vez que el estudiante elige dicha respuesta. El concepto se ejemplifica en la Figura 3.2.2.

DTD

```
<!ELEMENT Tutorial (Problem+)>
<!ELEMENT Problem (QuestionPoint | ((Text|Image)+,QuestionPoint?))>
<!ATTLIST Problem id ID #REQUIRED>
<!ELEMENT Text (#PCDATA)>
<!ATTLIST Text delay NMTOKEN "1">
<!ELEMENT Image (#PCDATA)>
<!ATTLIST Image path CDATA #REQUIRED delay NMTOKEN "1">
<!ELEMENT QuestionPoint (Answer*,AnotherAnswer)>
<!ELEMENT Answer (Response, Feedback+)>
<!ELEMENT AnotherAnswer (Feedback+)>
<!ELEMENT Response (#PCDATA)>
<!ELEMENT RespText (#PCDATA)>
<!ELEMENT Feedback ((Text|Image)+)>
<!ATTLIST Feedback next IDREF #IMPLIED>
```

Instancia XML

```
<!DOCTYPE Tutorial SYSTEM "tutorialV1.dtd">
<Tutorial>
  <Problem id="p1">
    <Text>7 + 5 suman...</Text>
    <QuestionPoint>
      <Answer>
        <Response>12</Response>
        <Feedback next="p2">
          <Image path="imgs/happy.jpg"/>
          <Text delay="2000">¡Correcto! Siguiente problema...</Text>
        </Feedback>
      </Answer>
      <Answer>
        <Response>35</Response>
        <Feedback next="p1">
          <Text>Fíjese bien en el símbolo de operación: +.</Text>
          <Text delay="2000">Inténtelo de nuevo.</Text>
        </Feedback>
        <Feedback next="p2">
          <Image path="imgs/sad.jpg"/>
          <Text>La operación es suma, no multiplicación.</Text>
          <Text delay="2000">La respuesta es 12. Siguiente problema...</Text>
        </Feedback>
      </Answer>
      <AnotherAnswer>
        <Feedback next="p1">
          <Text>No es correcto.</Text>
        </Feedback>
        <Feedback next="p2">
          <Text>La respuesta es 12. Siguiente problema...</Text>
        </Feedback>
      </AnotherAnswer>
    </QuestionPoint>
  </Problem>
  <Problem id="p2">
    <Text>Problema 2: ...</Text>
  </Problem>
</Tutorial>
```

Figura 3.2.1. Versión inicial de la gramática documental (DTD) de <e-Tutor> y ejemplo de tutorial (instancia XML) correspondiente a la Figura 3.2.2.

La motivación subyacente que dio lugar a la primera versión de <e-Tutor> fue el interés por experimentar con las técnicas dirigidas por lenguajes, especialmente el uso de lenguajes específicos del dominio basados en marcado, para el desarrollo de herramientas e-learning. Debido a las bondades, amplio uso y extensión que presenta XML, éste fue el lenguaje elegido para describir los tutoriales socráticos. La Figura 3.2.1 muestra la versión inicial de la DTD de

<e-Tutor> y un ejemplo de instancia XML, el cual establece el marcado para la descripción de los problemas de los documentos tutoriales: problemas que empiezan con (una posible secuencia vacía de) elementos de un tutorial, seguido de un punto de pregunta en el cuál el estudiante es interrogado.

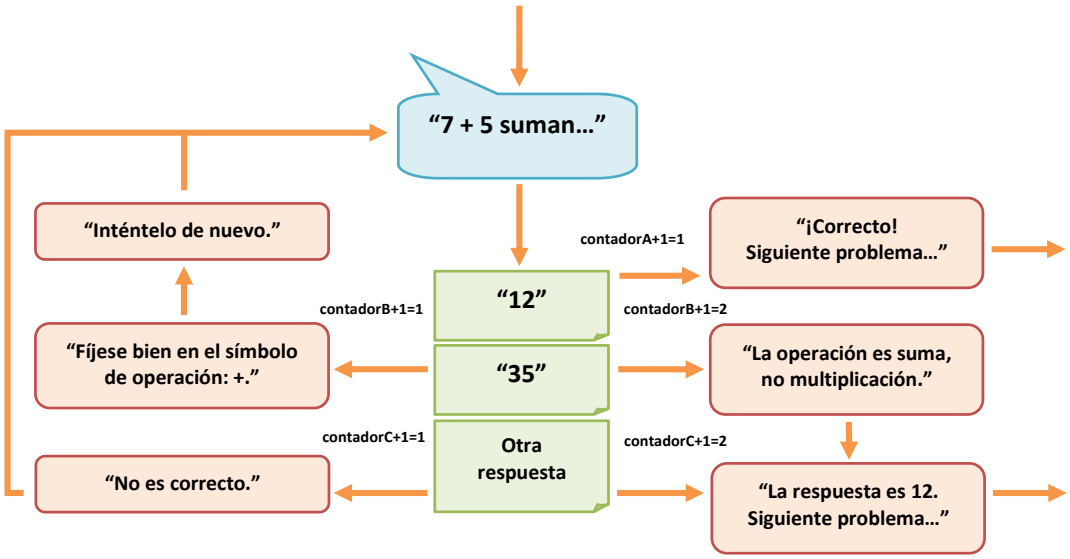


Figura 3.2.2. Ejemplo de un problema de un tutorial socrático.

En la versión inicial, la información mostrada al estudiante, como punto de partida, era la más básica: texto e imágenes como elementos de un tutorial. Como interacciones y forma de respuesta de parte del estudiante, se contempló utilizar la entrada de texto: el estudiante debía introducir una sentencia como respuesta a la pregunta planteada. La interfaz diseñada, por otro lado, era muy rudimentaria pero funcional. La Figura 3.2.3 muestra los principales componentes del marco de aplicación <e-Tutor> y la interfaz de usuario inicial.

Como un primer comienzo en el desarrollo del traductor, se comenzó por la codificación a mano del traductor directamente en Java, con el soporte de una API estándar de procesamiento XML basada en DOM. Aunque el desarrollo era viable, a medida que el lenguaje se volvía más complejo, este método indisciplinado de desarrollo se volvió insostenible. Este problema nos motivó a utilizar meta-herramientas para automatizar la generación del traductor. Nuestra primera solución se basó en un marco de aplicación Java para el desarrollo de generadores [Sierra et al. 2008]. Seguidamente utilizamos esquemas de traducción y herramientas asociadas. El enfoque resultante se describe en la sección 3.3. Finalmente, modificamos e hicimos evolucionar incrementalmente ambos enfoques, que confluyeron en el entorno XLOP 1.0, utilizado finalmente para el desarrollo con carácter profesional de <e-Tutor>, tal y como se mostrará en la sección 3.4.

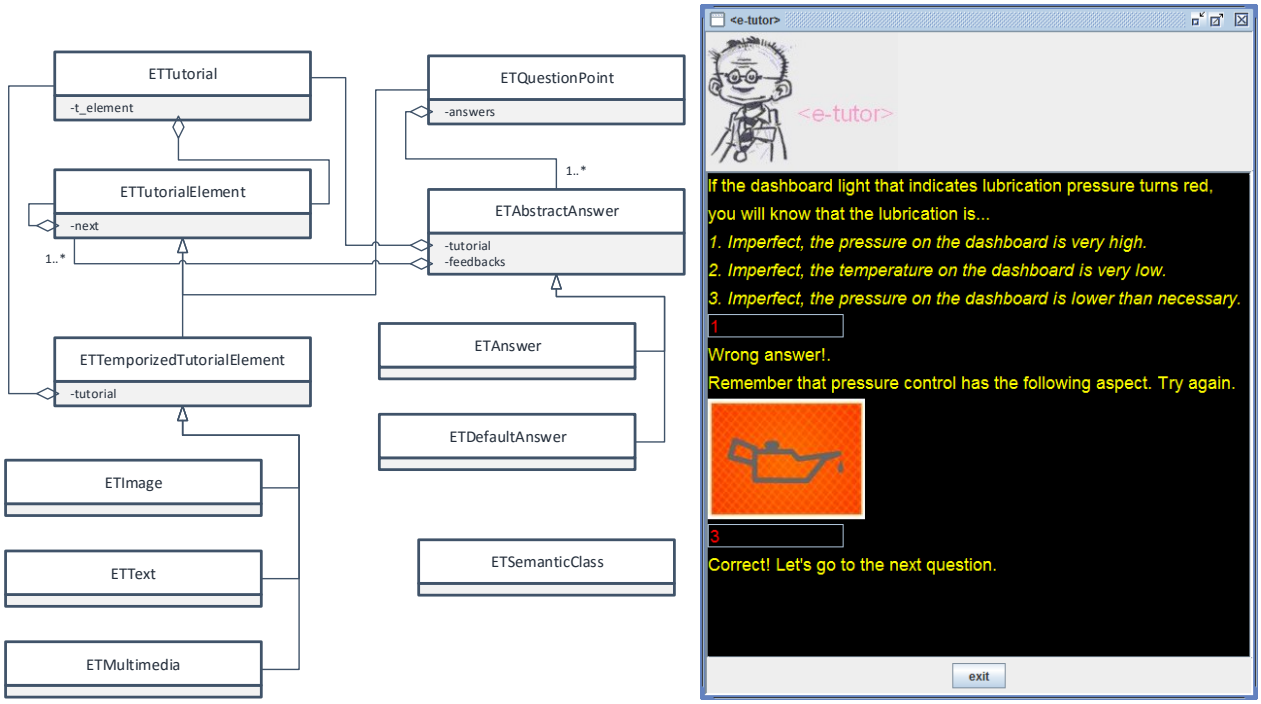


Figura 3.2.3. Marco de aplicación para representar tutoriales en <e-Tutor> (izquierda) e interfaz de usuario inicial de <e-Tutor> (derecha).

3.3 Desarrollo basado en meta-herramientas convencionales de construcción de procesadores de lenguaje

En esta sección se describe el enfoque de desarrollo planteado en [Sarasa et al. 2012] para la construcción de aplicaciones de procesamiento XML mediante esquemas de traducción. Para ello, el enfoque propone una metodología de desarrollo de carácter iterativo e incremental, dividida en la realización de cuatro actividades (sección 3.3.1). La primera actividad, denominada *Configuración del entorno de desarrollo*, consiste en establecer cómo se reconocen e interpretan los elementos de un documento XML mediante su análisis, actividad que se detallará en la sección 3.3.2. La segunda actividad, *Escritura del esquema de traducción*, consiste en la elaboración de una gramática incontextual con esquemas de traducción para describir el procesamiento a realizar sobre los elementos XML reconocidos, y se detallará en la sección 3.3.3. La tercera actividad, *Desarrollo de la lógica específica*, detallada en la sección 3.3.4, consiste en la elaboración del marco de la aplicación, cuyo comportamiento podrá ser modificado por las instrucciones presentes en los esquemas de traducción. La cuarta actividad, *Producción y prueba del procesador*, consiste en la generación de un procesador y traductor XML en base a las anteriores consideraciones, y se presenta en la sección 3.3.5. A lo largo de estas secciones se mostrarán detalles del desarrollo del caso de estudio <e-Tutor> mediante dicho enfoque de desarrollo, presentando finalmente en la sección 3.3.6, un caso de estudio de desarrollo de un entorno basado en el enfoque para facilitar y automatizar el proceso de desarrollo de las aplicaciones de procesamiento XML, precursor del entorno XLOP 1.0 [Sarasa et al. 2009-a].

3.3.1 Enfoque de desarrollo

En esta sección se presenta nuestro enfoque diseñado para el desarrollo de aplicaciones de procesamiento XML mediante herramientas convencionales de generación de traductores (YACC, CUP, etc., véase el Capítulo 2). El enfoque se centra esencialmente en el desarrollo de una serie de actividades principales y en su secuenciación.

La Figura 3.3.1 resume el flujo de trabajo del enfoque y sus actividades. El carácter que presenta es iterativo e incremental. Obsérvese en dicha figura cómo las flechas punteadas regresivas realzan dicho carácter. Además, los cambios realizados en una actividad no implican modificaciones en actividades previas. Si se observa con más detenimiento el flujo de trabajo de la figura, se puede observar cómo refleja en gran medida el proceso que se realizaría durante el desarrollo de un procesador de lenguaje convencional utilizando esquemas de traducción como método básico de especificación. Efectivamente:

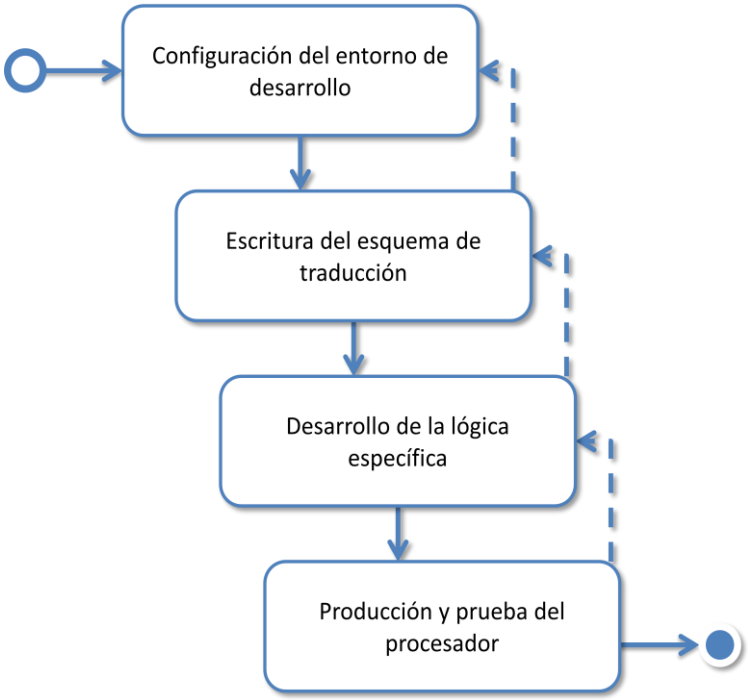


Figura 3.3.1. Secuencia de actividades del enfoque de desarrollo de aplicaciones de procesamiento XML mediante herramientas convencionales de generación de procesadores de lenguaje.

- Se comienza proporcionando una gramática incontextual que soporta el lenguaje a procesar y se continúa con el añadido de acciones semánticas a la gramática incontextual, con el fin de producir un esquema de traducción.
- Seguidamente puede implementarse la maquinaria necesaria invocada desde las acciones semánticas (p.e. en el caso de un traductor convencional, la tabla de símbolos, el módulo de gestión de errores, etc.).

- Finalmente, se podrá producir el traductor automáticamente mediante herramientas de generación de traductores como las ya mencionadas en el Capítulo 2.

La naturaleza de este proceso de desarrollo es, pues, dirigida por el lenguaje, pues su objetivo no es construir un componente de procesamiento de información de manera *ad hoc*, sino entender dicho desarrollo como el de un procesador de lenguaje. Para minimizar la complejidad, el enfoque propugna aportar una capa adicional de procesamiento de lenguaje sobre un marco de procesamiento XML, orientado a flujo, ya existente. En particular, el procesador operará sobre elementos de información XML en lugar de sobre caracteres individuales (por ejemplo, en la forma de representación aportada por SAX a través de sus eventos). Como resultado, la lógica específica de aplicación quedará interconectada y organizada a través de un marco de procesamiento general, dando lugar a la existencia de dos niveles bien diferenciados: un nivel que opera como un traductor dirigido por la sintaxis, y otro nivel que aporta servicios a dicho traductor. En las siguientes secciones se detallarán cada una de las actividades del enfoque de desarrollo introducidas en la Figura 3.3.1.

3.3.2 Configuración del entorno de desarrollo

La actividad *Configuración del entorno de desarrollo* integra una herramienta de generación de analizadores con un marco de procesamiento XML de propósito general. Esta actividad será realizada sólo esporádicamente, debido a que el mismo entorno de desarrollo puede ser usado en el desarrollo de diversas aplicaciones de procesamiento XML. Las herramientas de generación de analizadores, como las tratadas en el Capítulo 2, generan traductores para lenguajes formales a partir de especificaciones de alto nivel. Estos traductores están dirigidos por analizadores que operan sobre flujos de tokens aportados por analizadores léxicos. La idea clave subyacente, a la hora de realizar la integración, es visualizar los documentos XML como flujos de tokens. De esta manera:

- Dicha integración se basa en la siguiente estructura lógica: los flujos de tokens son producidos a partir de la información proporcionada por los marcos de procesamiento de propósito general, con lo que se evita operar directamente sobre los documentos XML.

- La integración en sí distingue entre cuatro categorías léxicas diferentes o tipos de tokens: (i) *tokens de datos* (fragmentos de contenido textual de los documentos procesados), (ii) tokens de etiquetas de apertura, (iii) tokens de etiquetas de cierre, y (iv) token de marca fin de documento.

Por su parte, cada token puede incluir información léxica adicional en forma de atributos léxicos. Más concretamente:

- Los tokens de datos poseen el contenido textual de información del propio documento.

- Los tokens de etiqueta de apertura contienen los atributos XML especificados en la etiqueta del documento procesado, así como el *espacio de nombres* (los espacios de nombres en XML son un mecanismo básico de modularización basado en cualificación de nombres) u otra información asociada a los mismos.

La Figura 3.3.2.a muestra un ejemplo de tokenización (reconocimiento y conversión de elementos de un documento en elementos de información manejables en el ámbito de la traducción). En base a dichas consideraciones, para la integración, se parte de un escáner XML genérico y configurable mediante el uso del marco de procesamiento XML deseado. Este componente puede ser genérico, puesto que sólo es necesario indicar cómo se realiza la asociación de etiquetas de apertura y cierre con las categorías léxicas correspondientes, por ejemplo, usando una tabla como la de la Figura 3.3.2.b. El componente en sí puede realizarse utilizando un marco de procesamiento orientado a flujos como SAX o StAX. De hecho, la acción de un escáner XML puede ser concebida como la transformación de un flujo de elementos de información de un documento en un flujo de tokens, formato de entrada esperado para los traductores generados.

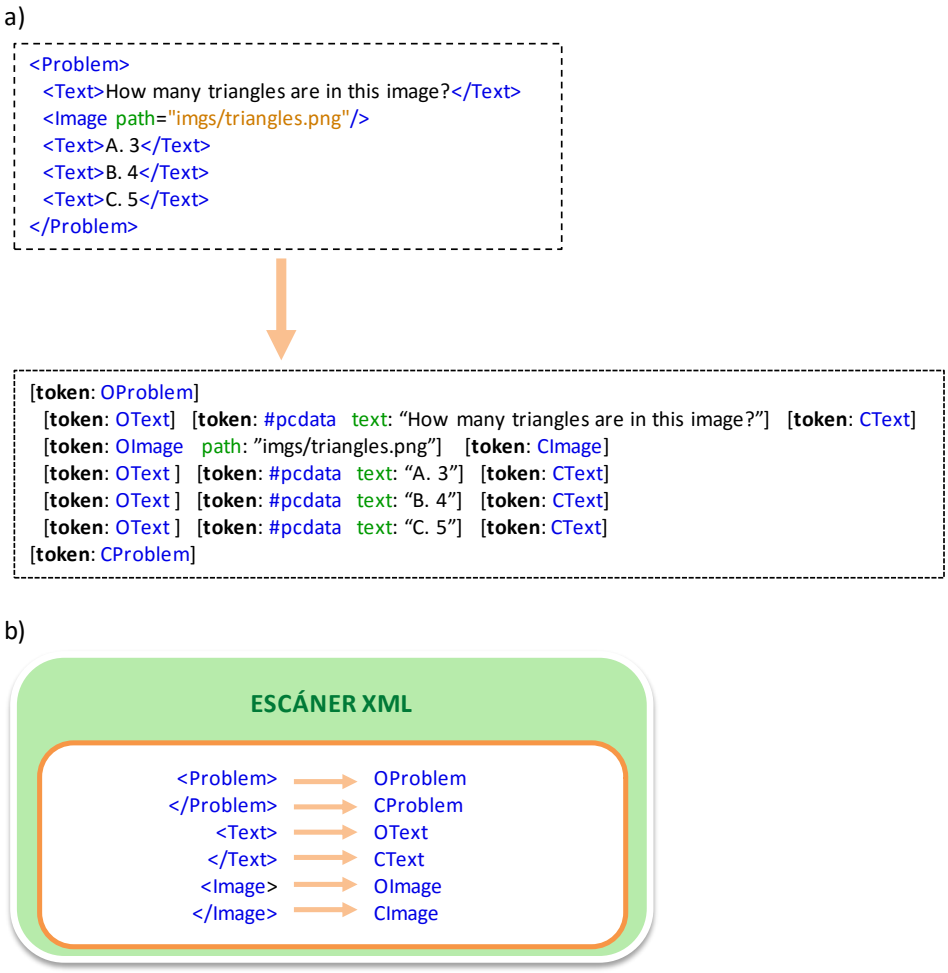


Figura 3.3.2. Ejemplo de tokenización (a) y configuración de un escáner XML (b).

Respecto a detalles técnicos, ya que los traductores generados son normalmente de tipo *push* (es decir, toman el control y solicitan los tokens a los escáneres cuando los requieren), la integración se vuelve especialmente sencilla con un marco de procesamiento XML de tipo *pull* (por ejemplo, StAX), debido a que estos marcos aportan cada elemento de información siguiente por demanda. La integración con un marco de tipo *push* (por ejemplo, SAX), requiere invertir el control (por ejemplo, usando una solución de tipo productor-consumidor multihilo). En [Sarasa et al. 2008, Sarasa et al. 2009-a] se muestran dos ejemplos de ambos tipos de integración.

Finalmente, cabe resaltar la diferencia entre el escáner XML propuesto y un escáner de un procesador de lenguaje convencional:

- El escáner XML propuesto en nuestro enfoque está construido encima de un marco de procesamiento XML orientado a flujo capaz de soportar características comunes a cualquier lenguaje de marcado XML, como son el soportar diferentes conjuntos de caracteres y codificaciones, comentarios, entidades y espacios de nombres entre otros.

- Sin embargo, el escáner de un procesador de lenguaje convencional suele funcionar sobre archivos de texto o flujos de caracteres.

Por tanto, aunque sería posible aportar un escáner convencional para convertir en tokens cualquier documento XML en particular, éste debe ser capaz de manejar las citadas características para dar soporte completo a los requisitos XML. La complejidad de los analizadores XML existentes nos muestra que esto no es una tarea fácil, resaltando la diferencia existente entre nuestro enfoque y el de desarrollo de un procesador de lenguaje convencional: si desarrollamos un procesador de lenguaje para un tipo de documento XML particular siguiendo patrones estándares explicados a un nivel de un curso de construcción de compiladores típico de una universidad, probablemente obtendríamos un programa capaz de procesar archivos de texto con un lenguaje parecido a XML, pero no un programa capaz de manejar las diferentes características comunes a todas las aplicaciones XML (por ejemplo, la capacidad de dividir un enorme documento XML en muchos archivos y poder unificar dichos fragmentos usando el mecanismo de entidades XML, o entre otros aspectos, la capacidad de soportar diferentes conjuntos de caracteres, espacios de nombres, etc.).

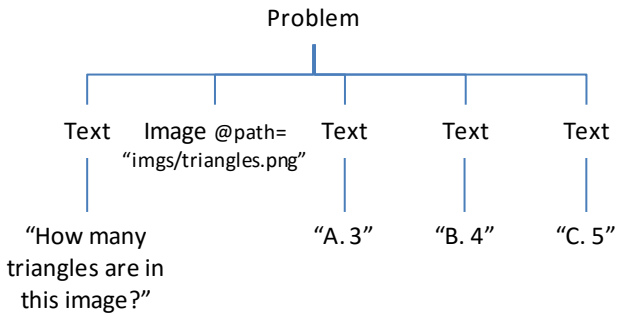
3.3.3 Escritura del esquema de traducción

La actividad *Escritura del esquema de traducción* constituye la parte central del enfoque de desarrollo que consiste en:

- Escribir una gramática específica para el procesamiento que defina una estructura adecuada para el flujo de tokens aportado por el escáner XML.

- Añadir acciones semánticas, mediante la escritura de código, a la gramática específica anterior, que describan el procesamiento en sí. El resultado es un esquema de traducción orientado al procesamiento y dirigido por la sintaxis.

a)



b)

```
Problems ::= Problems Problem | Problem
Problem ::= O_Problem ProblemDescription C_Problem
ProblemDescription ::= Element ProblemDescription | Element
Element ::= O_Text #pcdata C_Text | O_Image C_Image
```

c)

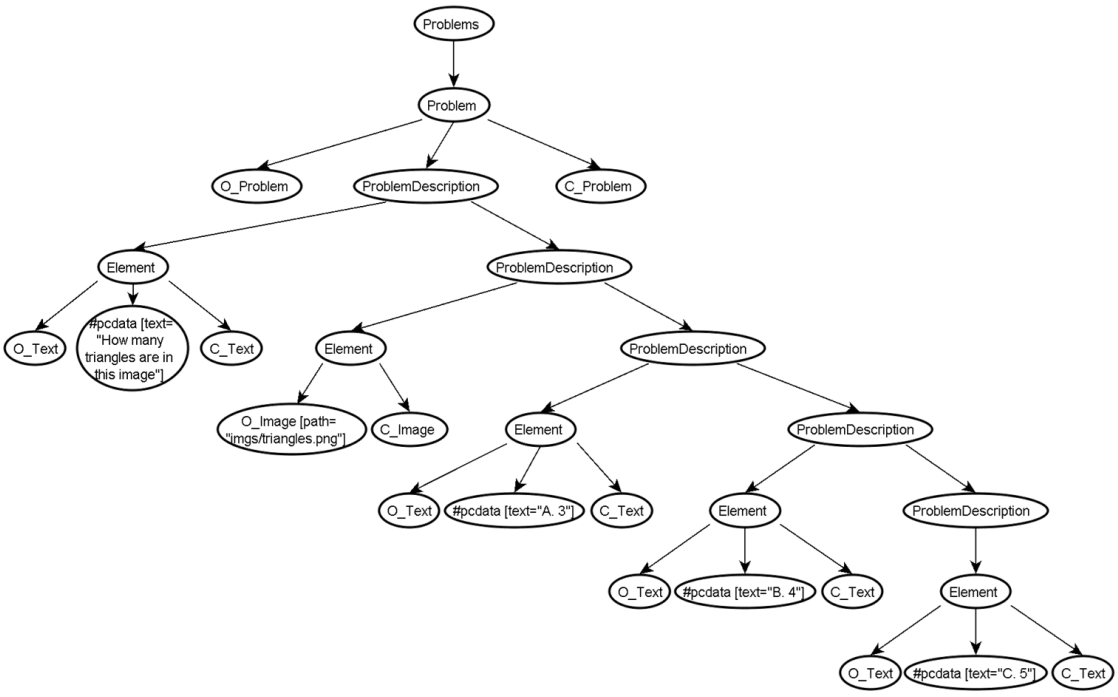


Figura 3.3.3. Árbol documental (a) para el documento de la Figura 3.3.2. Fragmento de gramática específica para el procesamiento (b). Árbol de análisis (c) para el documento de la Figura 3.3.2 respecto a la gramática de la Figura 3.3.3.b.

Es importante no confundir la gramática específica para el procesamiento con la gramática de definición del lenguaje de marcado del documento (DTD o esquema XML). La gramática específica de procesamiento de esta actividad aborda un aspecto clave en el proceso: dar una estructura apropiada al flujo de tokens para facilitar el procesamiento de los documentos. De hecho, este aspecto debe ser abordado por cualquier solución de procesamiento XML de propósito general. Por ejemplo, queda implícito en el código que maneja los hijos de un nodo elemento en una aplicación de procesamiento basada en DOM, en los métodos y variables de estado de un manejador de eventos SAX, o en el conjunto de procesos mutuamente recursivos de una aplicación basada en StAX (véase el Capítulo 2). La principal diferencia y ventaja de nuestro enfoque consiste en que la estructuración se describe explícitamente a un nivel de abstracción más alto, mediante una gramática incontextual, en vez de a nivel de código, en una implementación manual. La estructura impuesta para un flujo de tokens por una gramática específica de procesamiento adquiere la forma de un árbol de análisis. La Figura 3.3.3.b muestra un ejemplo, en donde puede observarse cómo el árbol de análisis adquiere una estructura más elaborada y precisa que el árbol documental (Figura 3.3.3.c), donde los contenidos adquieren una estructura más simple, uniforme y secuencial (Figura 3.3.3.a).

El modelo conceptual de procesamiento que subyace a un esquema de traducción orientado al procesamiento, es realizar un recorrido del árbol de análisis ejecutando las acciones semánticas en puntos significativos durante dicho recorrido. Además, las acciones semánticas pueden almacenar y consultar información en los nodos del árbol de análisis (típicamente esta información se organiza como una asignación de valores a atributos semánticos), o en variables globales. La naturaleza exacta de recorrido queda determinada por el tipo de traductores generados por la herramienta de generación de traductores (véase el Capítulo 2):

- Traductores descendentes, como los generados por JavaCC y ANTLR, que recorren el árbol de análisis en preorden (visitar nodos padres antes que sus nodos hijos). Los puntos significativos para ejecutar una acción semántica son, por cada nodo, cuando: (i) el traductor entra en el nodo, (ii) el traductor entra en un hijo, (iii) el traductor abandona a un hijo, y (iv) el traductor sale del nodo.

- Traductores ascendentes, como los generados por herramientas tipo YACC (por ejemplo, CUP), que recorren el árbol de análisis en postorden (visitar los nodos hijos de un nodo y después dicho nodo padre). Existe un punto significativo para ejecutar acciones semánticas cada vez que el traductor sale de un nodo.

Cabe destacar que, aunque es útil tener el modelo en mente al escribir los esquemas de traducción orientados a la generación, como ya se ha indicado en el Capítulo 2, únicamente es un modelo conceptual. En la práctica, el árbol de análisis no se construye, el recorrido se realiza implícitamente durante el proceso de análisis, y las acciones semánticas se ejecutan en un orden adecuado. Así mismo, los atributos semánticos están sólo disponibles como parámetros de procedimientos recursivos (por ejemplo, en traductores descendentes

recursivos generados por JavaCC o ANTLR) o almacenados en los registros de la pila semántica (por ejemplo, en un traductor ascendente generado mediante CUP). Este comportamiento es una característica fundamental cuando se trata con documentos muy grandes (como los que surgen en dominios *bigdata*) o con documentos disponibles asíncronamente como un flujo XML (como los requeridos en procesamientos on-line, en los que los documentos se procesan conforme estos se generan).

```
...
Problems ::= Problems Problem {
  $1.tutorialCreatedh = $$.tutorialCreatedh
  $2.tutorialCreatedh = $$.tutorialCreatedh
  $$.problemsCreated = ops.after($1.problemsCreated, ops.newProblem($2.id, $2.elem))
}

Problems ::= Problem {
  $1.tutorialCreatedh = $$.tutorialCreatedh
  $$.problemsCreated = ops.after($$.tutorialCreatedh, ops.newProblem($1.id, $1.elem))
}

Problem ::= <Problem> ProblemDescription </Problem> {
  $2.tutorialCreatedh = $$.tutorialCreatedh
  $$.id = $1.id
  $$.elem = $2.elem
}

ProblemDescription ::= Element ProblemDescription {
  $1.tutorialCreatedh = $$.tutorialCreatedh
  $1.tutorialCreatedh = $$.tutorialCreatedh
  $$.elem = ops.putNext($1.elem, $2.elem)
}

ProblemDescription ::= Element {
  $$.elem = ops.putNext($1.elem, ops.voidElem())
}

Element ::= <Text> #pcdata </Text> {
  $$.elem = ops.after($$.tutorialCreatedh, ops.newText($2.text))
}

Element ::= <Image> </Image> {
  $$.elem = ops.after($$.tutorialCreatedh, ops.newImage($1.path))
}
...
```

Figura 3.3.4. Extracto del esquema de traducción para el fragmento de gramática de la Figura 3.3.3.b perteneciente a <e-Tutor>.

Por otra parte, también cabe destacar que el modelo de traducción elegido puede suponer una limitación en el tipo de especificaciones que se pueden realizar, pues los traductores descendentes no funcionan con gramáticas recursivas a izquierdas, las cuales son útiles para caracterizar estructuras asociativas por la izquierda. Así mismo, aunque los traductores ascendentes están hechos para manejar recursión a izquierdas de manera eficiente, se vuelve substancialmente más difícil el manejo de información heredada (por ejemplo, información

que fluye desde un padre a un hijo o entre hermanos) respecto a los traductores descendentes [Aho et al. 2006].

Conociendo el recorrido que será realizado por el traductor, es posible colocar las acciones semánticas en las reglas sintácticas de la gramática específica de procesamiento. El formalismo de especificación debe aportar una manera de referir a los atributos semánticos (por ejemplo, colocándolos como parámetros de los símbolos sintácticos, como en JavaCC, o utilizando pseudovariables, como en herramientas de tipo YACC). La Figura 3.3.4 muestra un fragmento del esquema de traducción dirigido por la sintaxis de un traductor ascendente para <e-Tutor>. Se utiliza una notación similar a YACC (en particular, se utiliza pseudovariables de tipo YACC, $$ para referir a los registros semánticos de la cabeza de una regla, y $i para referir a los registros semánticos del i-ésimo símbolo del cuerpo siendo i > 0). El esquema de traducción construye una representación en memoria de los problemas de un tutorial de <e-Tutor>, seguido del típico patrón de generación de poblar un modelo semántico adecuado [Fowler 2010].

Finalmente, es importante resaltar que el enfoque se ha realizado de una manera lo suficientemente sencilla como para que las herramientas existentes de generación de traductores encajen adecuadamente. Por este motivo, las capacidades más avanzadas han sido explícitamente omitidas, aunque dichas capacidades permitan facilitar tareas de procesamiento más avanzadas. Por ejemplo, una de esas capacidades avanzadas podría ser la interacción entre sintaxis y semántica, soportada por herramientas como ANTLR [Parr 2007], las cuales, por ejemplo, permitirían realizar análisis en función de predicados sobre ciertos atributos semánticos. Aun así, sin embargo, es posible conseguir algunos comportamientos claves sin incluir estas características avanzadas, mediante la configuración del escáner XML para producir diferentes tokens para diferentes ocurrencias del mismo tipo de elemento, dependiendo de los valores de algunos de sus atributos XML.

3.3.4 Desarrollo de la lógica específica

Las acciones semánticas en un esquema de traducción típico utilizarán una maquinaria convencional, que debe también ser aportada para producir una aplicación de procesamiento XML completamente funcional. Esta maquinaria constituye la llamada *lógica específica de la aplicación*, y se proporciona durante la actividad de *Desarrollo de la lógica específica*. Por ejemplo, en <e-Tutor>, esta lógica específica de la aplicación está formada por el marco de aplicación <e-Tutor>, el cual constituye el ya mencionado modelo semántico en este escenario [Fowler 2010]. Como ya se ha mencionado anteriormente en la sección 3.3.3, la aplicación resultante instancia clases dentro del marco de aplicación <e-Tutor> (Figura 3.2.3) y vincula los objetos resultantes de manera adecuada (utilizando la terminología introducida en [Fowler 2010], puebla el marco de aplicación de <e-Tutor>). El enfoque fomenta una clara separación entre el procesamiento dirigidos por lenguajes de los documentos XML con el software convencional que soporta este procesamiento. Esta separación puede enfatizarse más mediante el aporte de una adecuada fachada para la lógica específica de la aplicación, con

operaciones que podrán ser invocadas desde el esquema de traducción (siguiendo el patrón descrito en [Fowler 2010]). En la Figura 3.3.4 puede observarse dicha práctica en <e-Tutor> mediante la variable global *ops*, que hace referencia a una instancia de dicha fachada representada por la clase semántica *ETSemanticClass* de la Figura 3.2.3.

3.3.5 Producción y prueba del procesador

En el momento en que el esquema de traducción y la lógica específica de aplicación estén disponibles, es posible obtener una aplicación funcional mediante el uso de una herramienta de generación de traductores. El proceso de producción, que se ilustra en la Figura 3.3.5, funciona de la siguiente manera:

- El esquema de traducción se usa como entrada para la herramienta de generación de traductores a fin de obtener una implementación de un traductor escrito en el lenguaje de salida de dicha herramienta (por ejemplo, Java para JavaCC o CUP). Observe que, de esta manera, la herramienta de generación se convierte en un tipo de *generador de aplicaciones* [Cleaveland 2001] en nuestra propuesta.

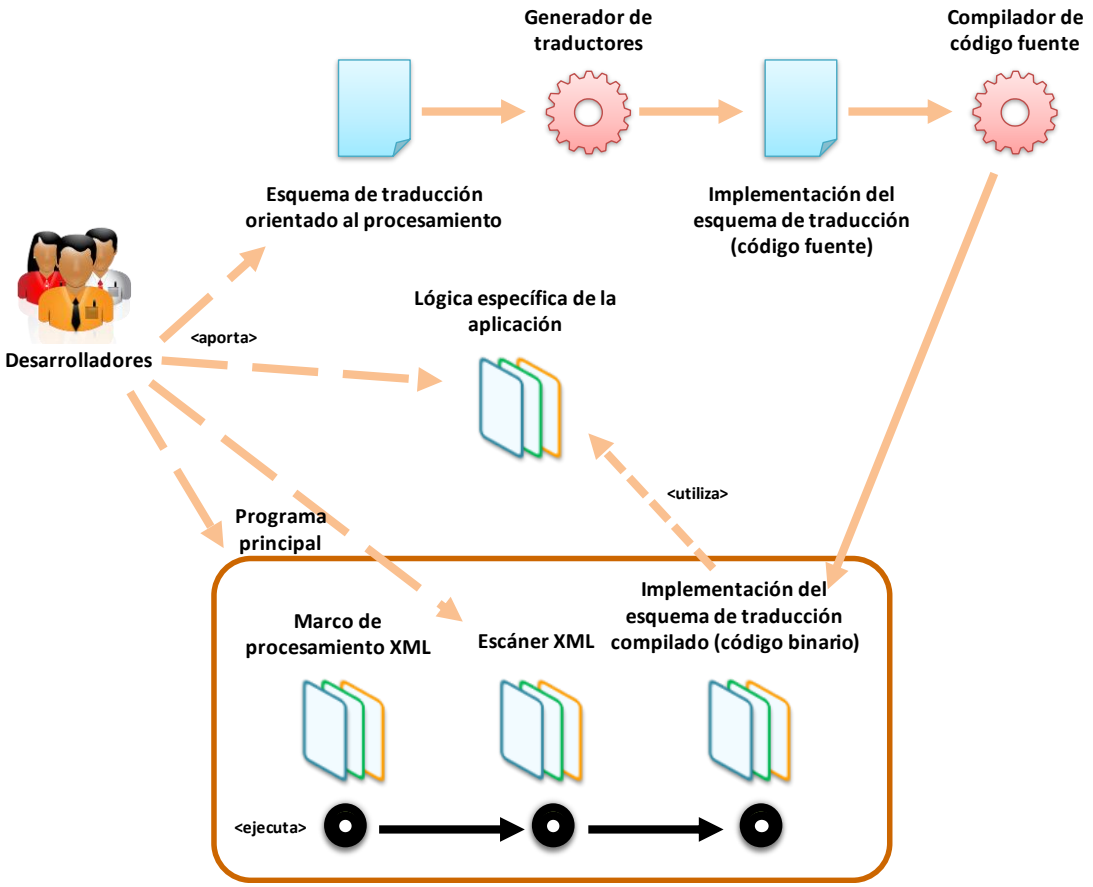


Figura 3.3.5. Proceso de producción de aplicaciones de procesamiento XML en nuestro enfoque de desarrollo.

- La implementación anteriormente obtenida, puede convertirse en un componente funcional utilizando un compilador para dicho lenguaje de salida (por ejemplo, un compilador Java, considerando el uso de JavaCC o CUP).

- También es necesario proporcionar el escáner XML debidamente configurado. Normalmente, sería necesario configurar la tabla de asociación para la tokenización (ver sección 3.3.2) utilizando un archivo de configuración, o directamente escribir dicha tabla en el lenguaje de salida (por ejemplo, Java).

- Por último, el desarrollador debe aportar un pequeño programa principal de unión. Este programa conectará apropiadamente todos los componentes requeridos que forman el flujo de procesamiento. Este flujo estará formado por: (i) un marco de procesamiento estándar XML capaz de transformar documentos XML en elementos de información (por ejemplo, los eventos representados por SAX) aptos para el escáner XML, (ii) el escáner XML personalizado utilizado para transformar dichos elementos en tokens aceptados por el traductor generado, y (iii) el traductor en sí, que hace uso de la lógica específica de la aplicación.

La aplicación resultante puede ser puesta a prueba a fin de resolver posibles defectos y errores en el funcionamiento. Con todo ello, esta actividad completa el proceso de desarrollo en nuestro enfoque.

3.3.6 Caso de estudio: entorno de desarrollo basado en CUP y StAX

Como caso de estudio, a lo largo de esta sección se presenta el diseño y desarrollo de un entorno para facilitar y automatizar el desarrollo de aplicaciones de procesamiento XML, elaborado a partir de herramientas generadoras de traductores y marcos convencionales de procesamiento XML. Dicho entorno se describe en [Sarasa et al. 2009-a]. Primeramente, se presenta una versión general de los componentes que fueron diseñados para la elaboración de un primer entorno, y cómo fueron mejorados para la elaboración de éste entorno, en la sección 3.3.6.1. Posteriormente, se detallará en la sección 3.3.6.2, cómo es la estructura de las aplicaciones generadas bajo dicho entorno. Para terminar, se mostrarán los últimos detalles de cómo se realiza la integración del marco de procesamiento XML en el entorno desarrollado, en la sección 3.3.6.3. A partir de este desarrollo, surge la primera versión del entorno XLOP: XLOP 1.0, que se detallará en la siguiente sección.

3.3.6.1 Visión general del enfoque

Como ya hemos visto anteriormente, las herramientas de generación de traductores automáticamente generan traductores partiendo de especificaciones gramaticales de alto nivel, las cuales pueden resultar útiles para construir aplicaciones de procesamiento XML. En [Sarasa et al. 2008] se muestra la combinación de la herramienta JavaCC para generar traductores escritos en Java, y una interfaz estándar SAX para el procesamiento de flujos XML basado en eventos. El resultado da lugar a un entorno de procesamiento XML dirigido por la sintaxis con las siguientes características:

- Para construir una aplicación de procesamiento XML, el entorno necesita la escritura de uno o más esquemas de traducción JavaCC. Todos estos esquemas de traducción deben ser construidos por encima de la misma gramática incontextual específica para el procesamiento. Esta gramática es una representación explícita orientada al procesamiento de la estructura de los documentos a ser procesados.

- El entorno usa JavaCC para transformar cada esquema de traducción en un traductor funcional escrito en Java.

- El entorno incluye un código en tiempo de ejecución capaz de conectar dichos traductores a cualquier analizador XML con soporte SAX. La conexión se lleva a cabo utilizando una tubería (*pipeline*) basada en SAX alimentada por una fuente de eventos SAX. El primer componente de la tubería tiene como finalidad realizar la asociación de eventos con los tokens esperados por el traductor. Los otros componentes distribuyen los tokens a dichos traductores que se ejecutan en hilos separados.

- Todos los traductores comparten una pila semántica específica. Cada entrada en dicha pila posee un contexto semántico adecuado al procesamiento de la tarea. Típicamente, este contexto expone un conjunto de atributos semánticos en términos de métodos *get* y *set*. También, estos métodos deben garantizar la coordinación de todos los traductores utilizando técnicas estándares de sincronización de hilos. Esta pila debe ser administrada como una pila semántica de un traductor descendente predictivo dirigido por tablas de análisis [Fischer et al. 2010]. Dicha administración debe ser explícitamente descrita por dos esquemas de traducción separados: uno para la creación de contextos y otro para su liberación.

La característica más potente de este entorno consiste en poder describir los diferentes aspectos de una aplicación de procesamiento XML a un alto nivel de abstracción, como un conjunto cooperativo de esquemas de traducción. Esta característica demostró ser muy útil durante la refactorización de complejas aplicaciones de procesamiento XML como <e-Tutor> y de un componente de validación de la biblioteca digital educativa Chasqui [Sierra et al. 2006]. No obstante, el entorno también exhibe múltiples deficiencias:

- El entorno restringe las gramáticas incontextuales específicas del procesamiento a la clase LL(1) (véase el Capítulo 2). Aunque JavaCC acepta gramáticas con uno o más símbolos de preanálisis, el ya mencionado mecanismo de coordinación basado en pila refuerza la restricción LL(1). Aunque esta no es una limitación seria, excluye algunas construcciones sintácticas que son más naturales para ciertas tareas de procesamiento.

- En un analizador descendente predictivo dirigido por tablas de análisis, el tamaño de la pila semántica es proporcional a la profundidad del árbol de análisis. De hecho, en el entorno propuesto en [Sarasa et al. 2008], el tamaño depende del tamaño del documento.

- También, debido a que la gramática específica de procesamiento son estrictamente de tipo LL(1) y a que JavaCC produce traductores descendentes recursivos, estos traductores también producen un número de llamadas recursivas encadenadas proporcional al tamaño del documento.

- Debido a que cada traductor corre en un hilo separado, la sincronización se vuelve pesada, especialmente en lo referente a la liberación de contextos semánticos: para liberar contextos semánticos se necesita verificar que el contexto no vaya a ser requerido por algún otro traductor de la cadena. Esta gestión debe ser especificada para cada nueva aplicación.

- Finalmente, aunque el entorno hace posible descomponer las tareas de procesamiento en etapas simples (uno para cada traductor), todas estas etapas se ejecutan mediante una simple pasada sobre árbol documental (en el orden del documento). Por lo tanto, el entorno no soporta procesos más complejos, que involucren múltiples pasadas en diferentes órdenes.

Para eliminar estas deficiencias preservando los beneficios del enfoque dirigido por la sintaxis, nuestro entorno se mejoró reemplazando JavaCC por CUP, y SAX por una interfaz StAX de tipo *pull* (véase el Capítulo 2), dando lugar al entorno descrito en [Sarasa et al. 2009-a]. A diferencia de JavaCC, como ya se ha indicado CUP genera traductores deterministas ascendentes. Además, acepta una clase más extensa de gramáticas: las LALR(1). También, bajo suposiciones razonables, los traductores generados son capaces de procesar largas cadenas de elementos de información con un tamaño de pila acotado. Puesto que los traductores generados típicamente por herramientas de generación de traductores tipo CUP son de carácter intrínseco *push*, encajan y funcionan por naturaleza muy bien con interfaces como StAX. De hecho, de esta manera, podemos evitar el uso de hilos de inversión de control, como en [Sarasa et al. 2008]. Por último, el nuevo entorno facilita la implementación de procesos complejos involucrando múltiples pasadas en el árbol documental. Para ello, permite utilizar un estilo dirigido por eventos para la programación de las acciones semánticas en el esquema de traducción, en vez de tener que depender de la solución multihilo adoptada en [Sarasa et al. 2008]. Esto simplifica dramáticamente el desarrollo de aplicaciones de procesamiento mientras se continúa evitando la construcción explícita de los árboles documentales.

3.3.6.2 Estructura de las aplicaciones

La Figura 3.3.6 presenta la estructura de las aplicaciones de procesamiento XML promovidas por el entorno. En ella, se muestra un componente predefinido y otros componentes específicos de aplicación:

- El componente predefinido en cuestión es el escáner XML. Este componente realiza la conexión con el marco de análisis StAX y produce los tokens requeridos por el traductor generado por CUP.

- En cuanto a los componentes específicos de la aplicación, estos se componen del traductor y la asociación de tokens. La asociación de tokens encapsula la lógica para transformar los elementos de información relevantes descubiertos por el analizador StAX en códigos para tokens requeridos por el traductor generado por CUP (tipos de tokens). Por su parte, el traductor implementa el procesamiento dirigido por la sintaxis del documento de entrada.

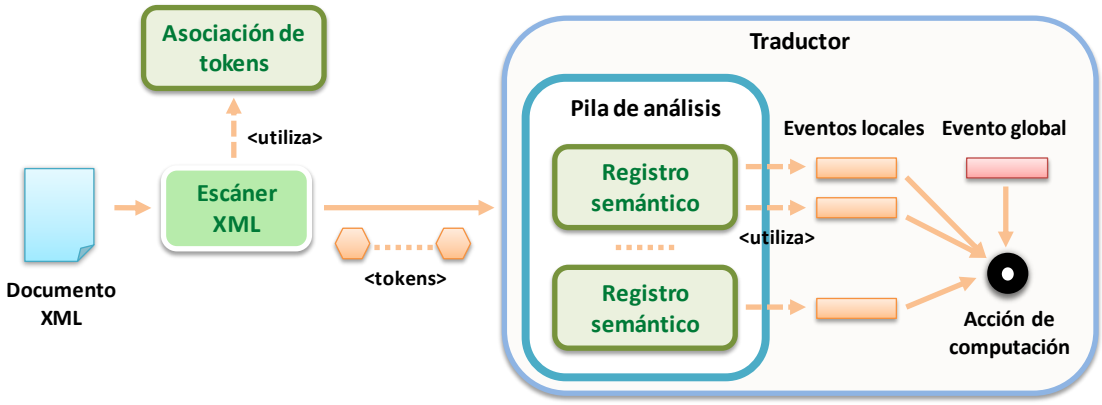


Figura 3.3.6. Estructura de las aplicaciones de procesamiento XML.

El traductor es un traductor ascendente que sigue el modelo de desplazamiento-reducción (véase el Capítulo 2). Mantiene, por tanto, una pila de análisis cuyas entradas corresponden con símbolos terminales y no terminales de una gramática incontextual específica para el procesamiento. Así mismo, opera sobre un flujo de tokens. Siguiendo el enfoque general, existe un token para cada etiqueta de apertura y cierre, así como por cada fragmento de texto y la marca fin de documento. El traductor realiza dos tipos diferentes de acciones al disponer de un token:

- Puede desplazar el token, que es apilado en la pila de análisis.

- Puede realizar una operación de reducción sobre una regla A ::= α (como se indica en el Capítulo 2, si la cadena α aparece en la cima de la pila de análisis, podrá ser reemplazada por A). De ser así, el traductor podrá ejecutar la acción semántica asociada a la regla A ::= α. Esta acción semántica puede hacer uso de variables

globales y actualizarlas, además de usar y actualizar los registros semánticos locales ligados a cada símbolo en la pila. Para símbolos terminales, estos registros contendrán la información relevante asociada a los tokens (elementos atributo para etiquetas de apertura, texto documental de fragmentos textuales). Para símbolos no terminales, el contenido de estos registros se deja al juicio del desarrollador.

En el entorno, los traductores son especificados a un alto nivel de abstracción, como esquemas de traducción CUP. Estos esquemas de traducción contienen las reglas sintácticas y sus acciones semánticas asociadas. CUP puede procesar estos esquemas para generar automáticamente los traductores. Como en [Sarasa et al. 2008], esto simplifica dramáticamente la producción y el mantenimiento de estos componentes. Adicionalmente, el traductor puede desacoplar algunos pasos de las acciones semánticas del flujo normal secuencial de un traductor ascendente. Para ello se introducen eventos y acciones de computación. Las acciones de computación dependen de eventos y sus activaciones. Estos eventos pueden ser locales o globales. Los eventos locales están asociados con registros semánticos, mientras que los globales están disponibles como variables globales. La Figura 3.3.6 ilustra estos aspectos.

3.3.6.3 Marco de integración

Como en [Sarasa et al. 2008], el entorno mejorado se compone de un marco de aplicación orientado a objetos Java ligero. En la Figura 3.3.7 se muestran las principales clases e interfaces de dicho marco. La clase *XMLScanner* implementa el ya mencionado escáner XML (la interfaz *Scanner* es la interfaz de CUP para analizadores sintácticos). Por lo tanto, funciona como puente a la maquinaria de StAX. A su vez, la interfaz *TokenMapper* especifica los métodos requeridos para la asociación de etiquetas de apertura y cierre, fragmentos de texto y marcas fin de documento, a tipos de tokens CUP. Consecuentemente, *XMLToken* y sus sub-clases permiten funcionar como puente entre los elementos de información XML y los tokens CUP (*Symbol* es la clase para estos tokens en CUP). Las clases *Event* y *CAction* son clases base para eventos y acciones de computación. Estas clases colaboran para producir un estilo de computación basado en eventos. La primera vez que se ejecuta una acción, se registran los eventos requeridos y después se inicia una ronda de ejecución. Durante esta ronda, la acción se encola en el primer evento requerido que todavía no haya sido disparado. Si no hay ninguno, entonces se ejecuta directamente la computación (a través de un método plantilla cuya implementación específica realiza la computación deseada). Cuando se dispara un evento, se invoca un método de ejecución para todas las acciones de computación encoladas en él.

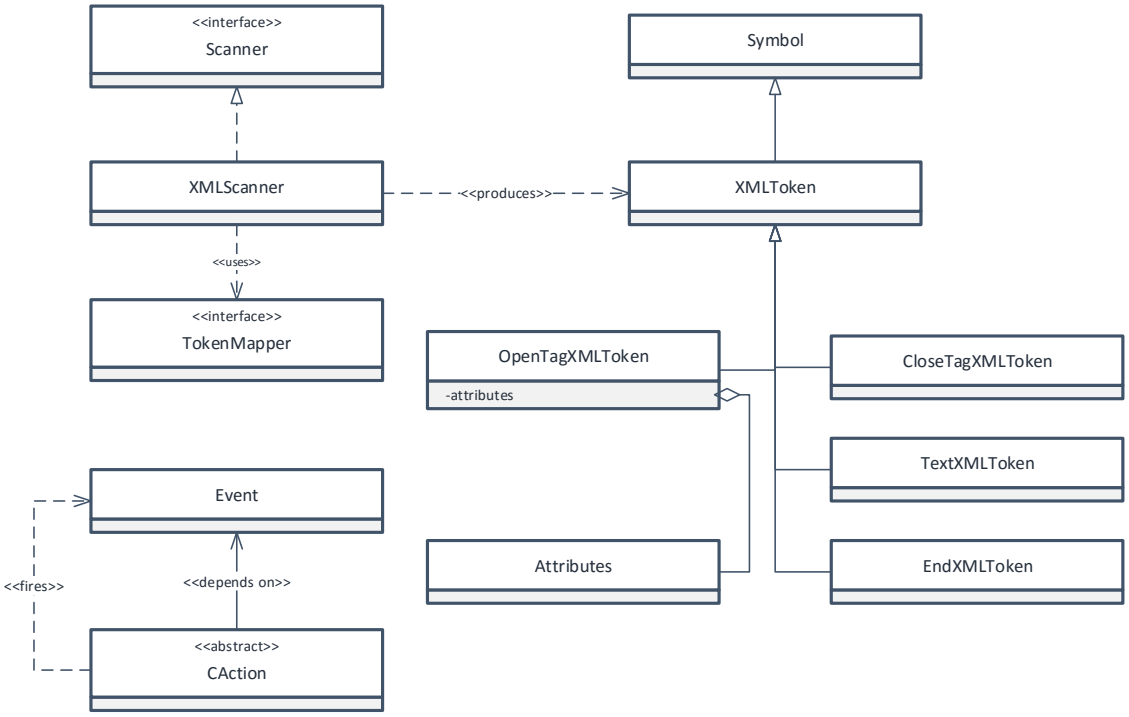


Figura 3.3.7. Clases principales e interfaces del entorno de procesamiento dirigido por sintaxis.

3.4 El entorno XLOP

XLOP [Sarasa et al. 2009-b, Sarasa et al. 2011] es un entorno de desarrollo de aplicaciones de procesamiento XML, que aporta una meta-meta-herramienta (en el sentido de que genera código para una meta-herramienta: CUP) que soporta gramáticas de atributos para facilitar y automatizar el desarrollo de dichas aplicaciones bajo el enfoque de desarrollo planteado en este capítulo. En [Temprado 2009] se describe una versión inicial de este entorno, y en [Temprado 2010] una versión exploratoria orientada al soporte de modularidad.

Utilizando este entorno, se muestra en la sección 3.4.1, cómo estas aplicaciones pueden desarrollarse con XLOP. Partiendo de una gramática de atributos descrita en el lenguaje de especificación XLOP, detallado en la sección 3.4.2, se mostrará, en la sección 3.4.3, cómo opera el generador del entorno para producir la aplicación de procesamiento XML final. Como consecuencia, el desarrollo se vuelve más sencillo, y como prueba, se presenta, en la sección 3.4.4, cómo se ha podido mantener y evolucionar, de una manera sencilla, la aplicación <e-Tutor> hasta dotarlo de un carácter profesional, gracias al entorno XLOP.

3.4.1 Visión general del entorno

XLOP (XML *Language-Oriented Processing*) es una herramienta para el desarrollo de aplicaciones de procesamiento XML basada en gramáticas de atributos. La Figura 3.4.1 muestra el desarrollo de una aplicación de procesamiento XML con XLOP.

El desarrollo de aplicaciones de procesamiento XML con XLOP se caracteriza por:

- Los desarrolladores escriben la especificación que describe la tarea de procesamiento XML, que será llevada a cabo por la aplicación, con una gramática de atributos. Para dicho propósito, se utiliza el lenguaje de especificación de XLOP.

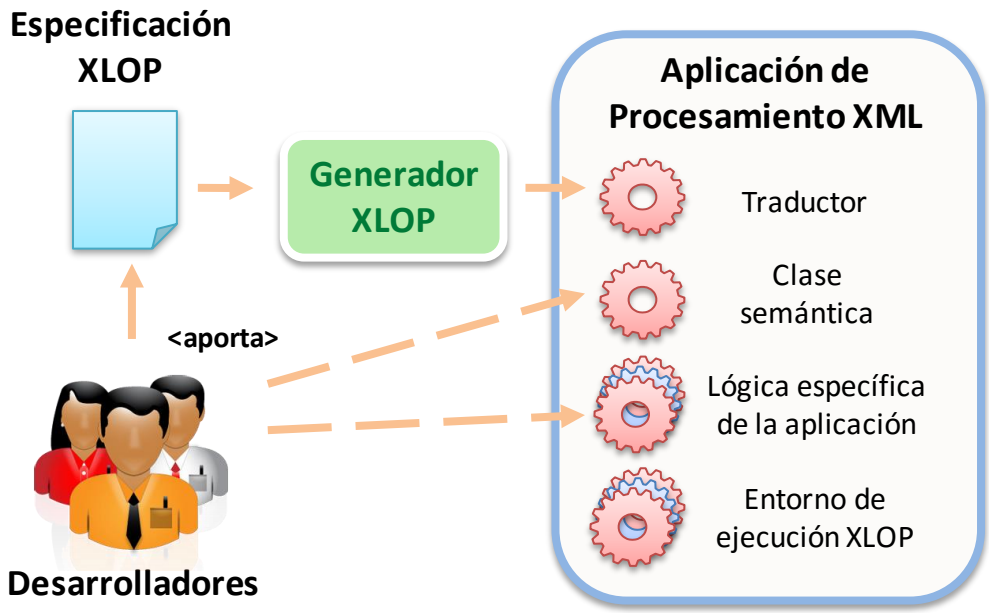


Figura 3.4.1. Desarrollo de aplicaciones de procesamiento XML con XLOP.

- Esta especificación es procesada por una herramienta llamada *Generador* XLOP. El resultado es un traductor funcional, componente principal que dirige el procesamiento de los documentos.

- Para poder operar, el traductor requiere una implementación de las funciones semánticas utilizadas en la especificación XLOP. Los desarrolladores deben aportar estas funciones como métodos de una *clase semántica* Java. Dichas funciones semánticas son útiles para conectar la aplicación con un entorno de desarrollo más amplio.

- Típicamente, las funciones semánticas harán uso de clases adicionales específicas de aplicación. Los desarrolladores deben aportar estas clases adicionales, que constituyen la denominada *lógica específica de aplicación*.

- La aplicación en sí resulta de unir el generador, la clase semántica, la lógica específica de aplicación y el entorno de ejecución de XLOP, el cual incluye una serie de componentes utilizados por el código generado. Entre estos componentes, se incluye uno que conecta con el analizador XML orientado a flujo subyacente, soportado por SAX (versión XLOP inicial) o por StAX (versión XLOP posterior).

De esta forma, la potencia de XLOP reside en que explícitamente promueve la separación del procesamiento XML en dos capas bien diferenciadas:

- Una capa formada por la lógica específica de aplicación, que agrupa toda la maquinaria necesaria para el soporte de la funcionalidad específica de la aplicación.
- Una capa lingüística, que se encarga del procesamiento dirigido por la sintaxis de los documentos XML. Esta capa se especifica utilizando una gramática de atributos.

La conexión entre las dos capas está dirigida por las funciones semánticas utilizadas en la gramática de atributos. Es importante remarcar que en el lenguaje XLOP, estas funciones semánticas no están realmente definidas en XLOP, sino que deben ser definidas externamente en Java, como métodos públicos de la clase semántica. De esta forma, la clase semántica actúa como una fachada para toda la maquinaria requerida para realizar la traducción.

3.4.2 El lenguaje de especificación

La sintaxis del lenguaje de especificación XLOP se muestra en Figura 3.4.2 (en [Sarasa et al. 2011] se describe una extensión de esta sintaxis orientada a facilitar la descripción de servicios web) Las gramáticas de atributos en XLOP son construcciones sobre gramáticas incontextuales especificas del procesamiento que representan la estructura lógica de un tipo de documento XML. Estas gramáticas pueden incluir los siguientes tipos de símbolos terminales:

- #pcdata. Denota un fragmento de contenido textual del documento procesado.

```
XLOPSpecification ::= {Rule}+
Rule ::= NonTerminal '::=' {SintacticElement}* '{' {Equation}* '}'
SintacticElement ::= NonTerminal | '#pcdata' | XMLElement
XMLElement ::= OpeningTag {SintacticElement}* ClosingTag |  EmptyElementTag
Equation ::= AttributeRef '=' SemanticExpression
SemanticExpression ::= Function '(' (SemanticExpression {',' SemanticExpression}*)? ')' | AttributeReference
AttributeReference ::= Attribute 'of' (NonTerminal | '#pcdata' | OpeningTag) ( '(' OccurrenceNumber ')' )?
```

Figura 3.4.2. Lenguaje de especificación XLOP.

- Etiquetas de apertura como <Problem>, <Image>, etc., y etiquetas de cierre como </Problem>, </Image>, etc. Estas etiquetas dependerán del lenguaje particular que

se procesa. Además, estas etiquetas deberán estar correctamente anidadas y emparejadas.

Adicionalmente, las gramáticas pueden utilizar símbolos no terminales. Con respecto a los atributos semánticos, el símbolo terminal *#pcdata* posee un único atributo léxico: *text*. Su valor será el contenido textual del documento al que se refiere. Las etiquetas de apertura pueden poseer atributos léxicos. Estos atributos corresponderán a los atributos de dichos elementos XML, situados en la etiqueta de apertura, en forma de asociación nombre-valor, como son descritos en el documento. En la versión 1.0 de XLOP no se restringe qué atributos pueden referirse. Por lo tanto, es responsabilidad del diseñador utilizar nombres apropiados para los atributos de un elemento en la especificación. Los símbolos no terminales pueden tener atributos sintetizados y heredados arbitrarios. No es necesario declarar estos atributos explícitamente, pues los atributos sintetizados y heredados de cada no terminal serán inferidos a partir de las ecuaciones semánticas definidas. En este proceso, se detectarán y reportarán las potenciales inconsistencias (p.e. un atributo que es usado con funcionalidad sintetizada y heredada al mismo tiempo para un mismo símbolo no terminal). Finalmente, los atributos de un símbolo serán referenciados con la notación de la forma *a of sr*, donde *a* es el nombre del atributo y *sr* la referencia del símbolo seguido de, opcionalmente, un número de ocurrencia entre paréntesis. Este número de ocurrencia será necesario para aquellas producciones donde el mismo símbolo aparece una o más veces en su regla sintáctica. Así mismo, es importante indicar que en la versión 1.0 de XLOP, el uso apropiado de las funciones semánticas se detecta en tiempo de ejecución, por lo que el desarrollador debe verificar que dichas funciones existen y están definidas correctamente tanto en la clase semántica, como en la especificación XLOP.

```
...
Problems ::= Problems Problem {
  unlinkedTutorial of Problems(0) = addNewProblem(unlinkedTutorial of Problems(1), id of Problem, elem of Problem)
  unlinkedTutorial_inh of Problems(1) = unlinkedTutorial_inh of Problems(0)
  unlinkedTutorial_inh of Problem = finalTutorial of Problems(1)
  finalTutorial of Problems(0) = finalTutorial of Problem
}
...
```

Figura 3.4.3. Extracto de especificación XLOP para el traductor de <e-Tutor>.

La Figura 3.4.3 muestra un extracto de especificación XLOP. Las tareas de procesamiento que describe consisten en el instanciamiento y la vinculación adecuada del marco de aplicación orientado a objetos dedicado a la representación de tutoriales socráticos (Figura 3.2.3). Estas tareas consisten en crear cada pieza del tutorial en una primera pasada, y vincular las piezas entre sí en una segunda pasada. Mientras que el orden de los diferentes pasos en cada pasada es irrelevante, la segunda pasada no puede comenzar hasta que la primera haya terminado. Por lo tanto, los pasos en la segunda pasada dependen de un evento apropiado de construcción. Es por ello que en la Figura 3.4.3 se ve reflejado cómo primeramente deben crearse los elementos del tutorial y posteriormente, propagar la información por el árbol de derivación implícito mediante atributos heredados/sintetizados para la vinculación de los elementos del tutorial entre sí.

3.4.3 El generador

XLOP incluye un generador que, tomando como entrada una gramática de atributos descrita en el lenguaje de especificación XLOP, produce una especificación CUP como salida. La especificación CUP generada puede ser compilada por CUP para producir una implementación Java perteneciente a la aplicación de procesamiento XML final. El código generado utiliza diversas clases que constituyen el entorno de ejecución de XLOP. Este entorno incluye un componente que actúa como un escáner para el traductor generado, el cual conecta con el marco de análisis XML basado en el estándar SAX. Este componente, como también la estrategia de interconexión, es análoga a lo que se describe en [Sarasa et al. 2008]. La Figura 3.4.4 muestra el proceso de generación en XLOP.

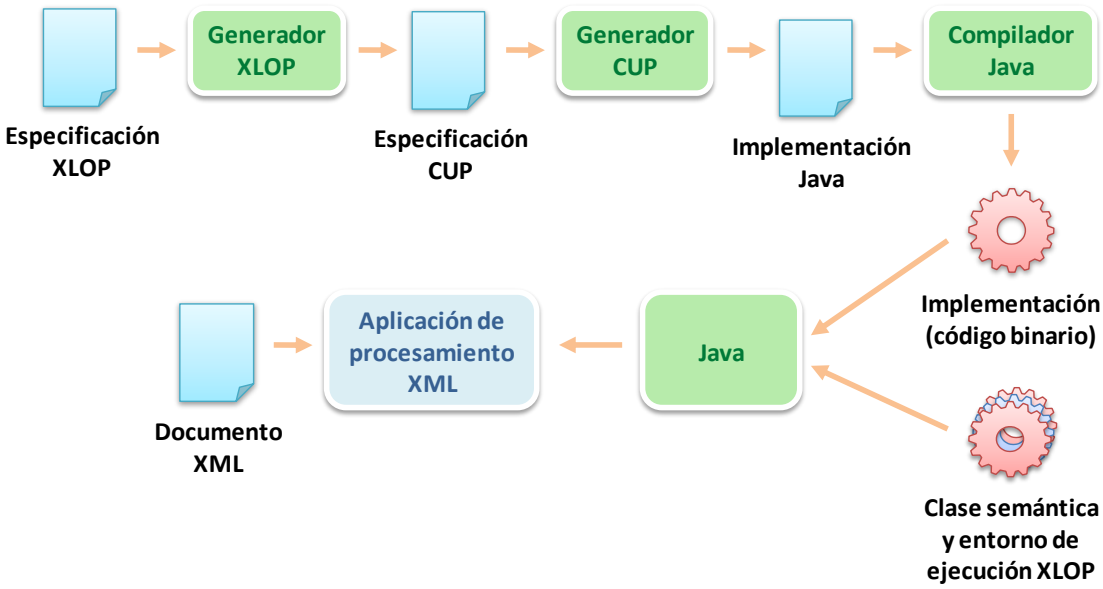


Figura 3.4.4. Proceso de generación en XLOP.

Como anteriormente hemos visto, CUP es un sistema de generación de analizadores en Java que soporta gramáticas LALR(1). Por lo tanto, XLOP soporta satisfactoriamente gramáticas incontextuales específicas para el procesamiento de carácter LALR(1). También, XLOP es capaz de manejar atribuciones no circulares arbitrarias de este tipo de gramáticas LALR(1). Esto se consigue mediante el uso de un patrón de ejecución retardada. De hecho, el valor semántico de cada símbolo sintáctico en la especificación CUP generada pertenece a una clase con una variable miembro pública por atributo. Estas variables son de la clase *Attribute*, una clase que ofrece tres métodos principales: un método de lectura de atributos (*getValue*), un método de escritura (*setValue*), y un método de retraso (*whenAvaliable*). Este último método toma una *continuación* como entrada: un objeto que implementa la interfaz *Continuation*, donde se declara el método *next*. Si el valor del atributo es conocido, se invoca la continuación. En caso contrario, encola la continuación hasta que dicho valor está disponible. A su vez, *setValue* invoca todas las continuaciones pendientes. La clase *Attribute* y la interfaz *Continuation* se muestran en la Figura 3.4.6.

```
 ...
Element ::= _O_Image: _O_Image0 _C_Image
{:
  RESULT = new SemanticElement();   final SemanticElement a0 = RESULT;   final Attributes a1 = _O_Image0;
  a0.tutorialCreatedh.whenAvaliable (
    new Continuation() {
      public void next() {
        a0.elem.set(XLOPHelper.invoke(semObj, "after", new Object[]{
          a0.tutorialCreatedh.get(), XLOPHelper.invoke(semObj, "newImage", new Object []{a1.getValue("path")})
   }));}});
:};
 ...
```

Figura 3.4.5. Fragmento de especificación CUP generado por XLOP a partir del extracto de especificación XLOP de <e-Tutor>.

Utilizando el patrón de ejecución retardada, es posible embeber el proceso de evaluación dirigida por dependencias fácilmente en el proceso secuencial de análisis llevado a cabo por el analizador generado por CUP. La Figura 3.4.5 muestra un fragmento de la especificación CUP generada para el ejemplo de aplicación <e-Tutor>.

Attribute.java

```
import java.util.ArrayList;

public class Attribute<T> {

  private T value;
  private ArrayList<Continuation> operations;
  private boolean available;

  public Attribute() {
    value = null;
    operations = new ArrayList<Continuation>();
    available = false;
  }

  public T get() {
    return value;
  }

  public void set(T value) {
    this.value = value;
    available = true;
    for (int i = 0; i < operations.size(); i++) {
      operations.get(i).next();
    }
  }

  public void whenAvaliable(Continuation aux) {
    if (available) aux.next();
    else this.operations.add(aux);
  }
}
```

Continuation.java

```
public interface Continuation {
  void next();
}
```

Figura 3.4.6. Clase Attribute e interfaz Continuation.

3.4.4 Caso de estudio: mantenimiento y evolución de <e-Tutor>

La herramienta <e-Tutor> ha ido mejorando y evolucionando en iteraciones sucesivas para incluir algunas mejoras en la interfaz de usuario y en la lógica interna de reproducción, como puede verse en la Figura 3.4.7. Como se describe en [Temprado et al. 2010-b], estos cambios en la herramienta se han llevado a cabo sin cambiar la especificación XLOP, como una consecuencia indirecta de la separación de aspectos promovidos por las gramáticas de atributos y XLOP. De hecho, las funciones semánticas, y por lo tanto la clase semántica, constituyen una clara interfaz definida entre el traductor y el marco de trabajo que soporta a la herramienta, que interconecta la capa lingüística, manejada por el lenguaje de especificación XLOP, con la capa de la lógica específica de aplicación, manejada por métodos de desarrollo de software convencional. Gracias a ello, a los desarrolladores se les permite realizar cambios en el marco de aplicación durante el desarrollo sin requerir cambios adicionales en la especificación XLOP, siempre y cuando no existan nuevas características que hayan sido introducidas en el lenguaje del dominio de la aplicación (en este caso, descritas por la DTD de los documentos tutoriales).

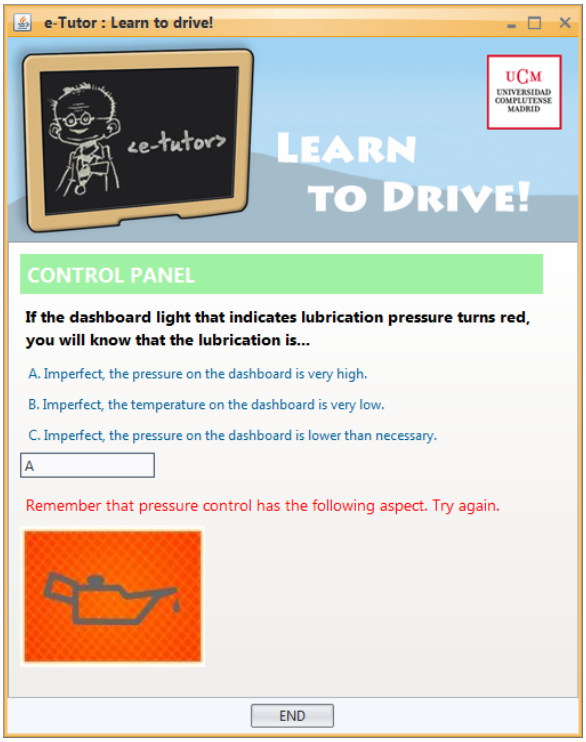


Figura 3.4.7. Interfaz de usuario mejorada de <e-Tutor>.

Por otro lado, durante el desarrollo de la herramienta, la especificación XLOP en sí fue refactorizada, simplificada y mejorada. Nos dimos cuenta que las instancias de la clase semántica, al ser una clase Java convencional, podían mantener estado (mediante el almacenamiento de valores en variables globales propias de la clase Java) como una característica adicional que simplificara la especificación, en concreto, disminuyendo los parámetros y el flujo de entrada y salida de datos entre las funciones semánticas. La

encapsulación del estado en las instancias nos permitió hacer que las principales estructuras fueran almacenadas en la clase semántica y se refactorizó la especificación XLOP para que los atributos mantuviesen dependencias entre cambios de estados en vez de entre valores. Esta técnica de especificación supuso una gran simplificación de la especificación original, evitando introducir ecuaciones semánticas para describir el flujo de atributos sintetizados/heredados cuyo propósito era meramente mantener el estado (véase le Figura 3.4.8 respecto la Figura 3.4.3). La modernización del traductor ha sido independiente, pues no implican cambios en el marco de la herramienta, gracias a la separación de aspectos que nos brindan las gramáticas de atributos.

```
...
Problems ::= Problems Problem {
  tutorialCreated_i of Problems(1) = tutorialCreated_i of Problems(0)
  tutorialCreated_i of Problem = tutorialCreated_i of Problems(0)
  problemsCreated of Problems(0) = after(problemsCreated of Problems(1), newProblem(id of Problem, elem of Problem))
 }
...
```

Figura 3.4.8. Extracto de especificación XLOP mejorada de <e-Tutor>.

Finalmente, para incrementar la riqueza de contenidos y dotar a <e-Tutor> de métodos de interacción como respuesta a preguntas más ricas, se ha extendido el lenguaje de <e-Tutor> para soportar selección de zonas dentro de imágenes, reproducción de formatos de videos, presentaciones Flash™ y streaming de videos de YouTube™, como puede verse en la Figura 3.4.9 y Figura 3.4.10).

DTD para los nuevos tutoriales de <e-Tutor>

```
...
<!ELEMENT Problem ((QuestionPoint | ImageQuestionPoint) |
((Text|Image|Multimedia|Flash|YouTube)+,(QuestionPoint|ImageQuestionPoint)?))>
<!ELEMENT Multimedia (#PCDATA)>
<!ATTLIST Multimedia path CDATA #REQUIRED>
<!ELEMENT Flash EMPTY>
<!ATTLIST Flash path CDATA #REQUIRED>
<!ELEMENT YouTube EMPTY>
<!ATTLIST YouTube path CDATA #REQUIRED>
<!ELEMENT ImageQuestionPoint (Answer+,AnotherAnswer)>
<!ATTLIST ImageQuestionPoint path CDATA #REQUIRED>
<!ELEMENT Feedback ((Text|Image|Multimedia|YouTube|Flash)+)>
...
```

Figura 3.4.9. Extensión del lenguaje de <e-Tutor> (extracto DTD) para el soporte de reproducción de vídeo, contenido Flash™, streaming de vídeo YouTube™, e imágenes interactivas.

La inclusión de estas nuevas funcionalidades, gracias al método de desarrollo implementado en XLOP, se vuelven tareas muy sencillas: por cada nueva funcionalidad implementada en el marco de aplicación <e-Tutor>, la extensión del traductor consiste simplemente en añadir una nueva regla en el lenguaje de especificación y, realizar (si procede), la conexión necesaria mediante nuevas funciones semánticas en la clase semántica.

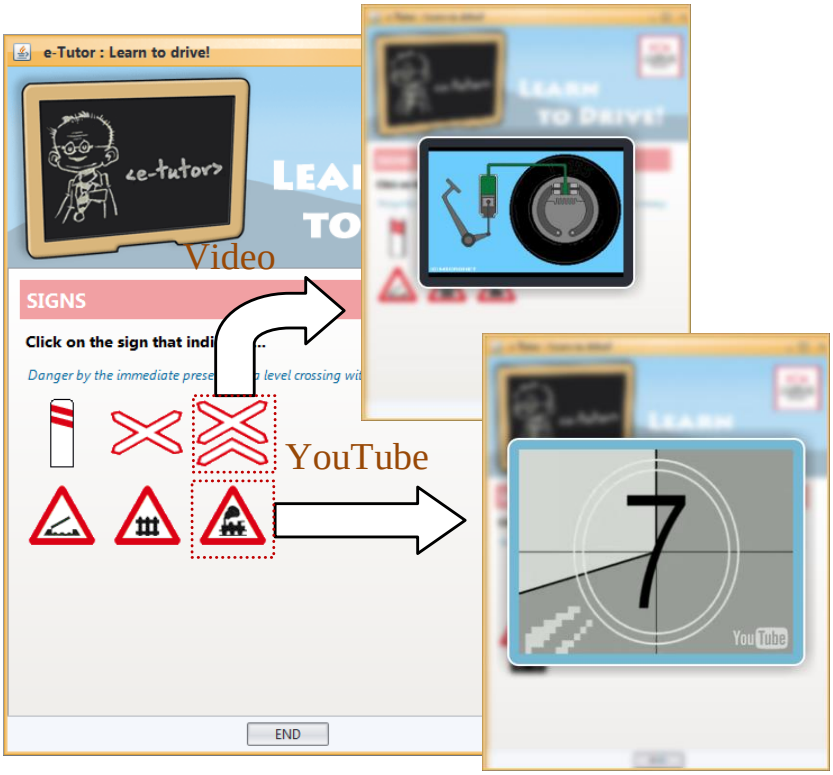


Figura 3.4.10. Nuevas funcionalidades multimedia añadidas a los tutoriales <e-Tutor>.

Para finalizar, en la Figura 3.4.11 se muestra un ejemplo para ilustrar lo sencillo que resulta el añadir una nueva funcionalidad (YouTube™) a <e-Tutor>. En este caso el método *newYouTube* debe implementarse en la clase semántica y realizar la instanciación de un elemento *ETYouTube*, clase y funcionalidad nueva ya implementada en el marco de aplicación <e-Tutor>.

```
...
Element ::= <YouTube/> {
  elem of Element = after(tutorialCreated_i of Element, newYouTube(path of <YouTube>))
}
...
```

Figura 3.4.11. Añadido de la funcionalidad YouTube™ a <e-Tutor>.

3.5 A modo de conclusión

Este capítulo ha analizado de manera unificada los distintos enfoques dirigidos por lenguajes para la construcción de aplicaciones XML que hemos desarrollado en el grupo de investigación ILSA. Básicamente, dichos enfoques se dividen en aquellos que proponen utilizar herramientas de generación de traductores convencionales (basados en esquemas de traducción), y el que propone utilizar gramáticas de atributos (y que ha llevado a la

formulación del sistema XLOP). No obstante, y a pesar de sus aparentes diferencias, todos ellos comparten unas características comunes que merece la pena enfatizar:

- Los enfoques propugnan la adaptación de los modelos de proceso clásicos utilizados en el desarrollo de un procesador de lenguaje para la construcción de aplicaciones de procesamiento de documentos XML. De hecho, todos ellos, independientemente de si se tratan de enfoques orientados a esquemas de traducción u orientados a gramáticas de atributos, fomentan actividades orientadas a la caracterización de la sintaxis o a la caracterización del procesamiento del lenguaje como ejes centrales del proceso de desarrollo.

- Todos ellos fomentan la formulación de una gramática incontextual específica para el procesamiento, que es *independiente* de la gramática documental. De esta forma, ambas gramáticas son artefactos distintos, con diferente propósito. Adoptando una analogía con el desarrollo de un compilador para un lenguaje de programación, mientras que la gramática documental sería el análogo a los diagramas sintácticos o la especificación EBNF que describe la sintaxis en el manual de usuario del lenguaje, la gramática específica para el procesamiento es la gramática que sirve como base para construir el módulo de análisis sintáctico del compilador.

- Todos estos enfoques propugnan, así mismo, una estrategia de procesamiento dirigida por la sintaxis, en el sentido de promover el posterior añadido de semántica a la gramática específica para el procesamiento, ya sea mediante el añadido de acciones semánticas (en el caso del uso de esquemas de traducción y de meta-herramientas convencionales para el desarrollo de procesadores de lenguaje), ya sea mediante el añadido de atributos y ecuaciones semánticas (en el caso del formalismo de las gramáticas de atributos y de XLOP).

- Todos estos enfoques promueven, así mismo, una estructuración de las aplicaciones en dos capas bien diferenciadas: la capa *de la lógica específica de la aplicación* (que, normalmente, se plasma en bibliotecas o marcos de aplicación específicos para la aplicación), y la capa *lingüística*, que se encarga de llevar a cabo el procesamiento de los documentos, de acuerdo con un enfoque dirigido por la sintaxis específica para el procesamiento. La forma de conectar dichos niveles es a través de una *clase semántica*, que implementa las operaciones básicas que se invocan desde la capa lingüística en términos de las funcionalidades ofrecidas por la capa de la lógica específica de la aplicación.

- Todos los enfoques adoptan una estrategia generativa para construir, mantener y hacer evolucionar a la capa lingüística. Para ello propugnan utilizar formalismos específicos de alto nivel (esquemas de traducción, gramáticas de atributos) para caracterizar dicha capa, y generarla automáticamente mediante el uso de meta-herramientas adecuadas. Este hecho evidencia el carácter metalingüístico de estos enfoques (ya que propugnan el uso de especificaciones metalingüísticas a partir de las cuáles es posible generar las capas lingüísticas de las aplicaciones).

- Por último, todos los enfoques promueven integrar los procesadores generados a partir de las especificaciones de alto nivel con marcos de procesamiento XML de propósito general convencionales. La maquinaria de procesamiento XML convencional se utiliza, de esta forma, para proporcionar componentes análogos a los analizadores léxicos presentes en la arquitectura habitual de un procesador de lenguaje.

A este respecto, es interesante enfatizar cómo la separación explícita entre la capa lingüística y la de la lógica específica de la aplicación, junto con el enfoque generativo adoptado para el desarrollo de la capa lingüística, aportan a estos enfoques las ventajas de generalidad de las tecnologías de procesamiento de documentos XML de propósito general (DOM, SAX, StAX, etc.), ya que el enfoque no restringe la naturaleza de la capa de la lógica específica, y las ventajas de los enfoques específicos, ya que el desarrollo de la capa lingüística se lleva a cabo utilizando formalismos y notaciones específicas para el desarrollo de procesos dirigidos por la sintaxis. Así mismo, el uso de un componente mediador entre ambas capas muy bien delimitado, la clase semántica, posibilita, en algunos casos, patrones que facilitan el proceso de especificación declarativa. Este es el caso, por ejemplo, del uso de estado en la clase semántica en conjunción con las especificaciones XLOP, y el uso de atributos que, en lugar de contener valores, representan cambios de estado, y, por tanto, puntos específicos en la evolución del procesamiento.

Por último, también es interesante observar que, en los enfoques analizados:

- Por una parte, hemos ignorado la relación que existe entre las gramáticas específicas para el procesamiento y las correspondientes gramáticas documentales. Efectivamente, el que se traten de artefactos independientes y diseñados para cumplir propósitos diferentes no implica que no estén relacionados, ya que ambos artefactos modelizan, en último término, un mismo lenguaje. Por tanto, debe existir algún tipo de relación de equivalencia entre ambas descripciones (en el mismo sentido en el que, por ejemplo, la especificación EBNF de un lenguaje de programación debe generar el mismo lenguaje que, por ejemplo, la gramática BNF LALR(1) utilizada para construir el correspondiente analizador sintáctico). No obstante, el cumplimiento de dicha relación se ha relegado al buen juicio del desarrollador, sin haberse establecido criterios automatizables que garanticen dicho cumplimiento.

- Por otra parte, la mayor parte de los enfoques desarrollados en ILSA (y, desde luego, los analizados en este capítulo, en cuyo desarrollo ha intervenido activamente el autor de esta tesis) son monolíticos, en el sentido de que la capa lingüística se especifica mediante una especificación global (un único esquema de traducción, una única gramática de atributos). A este respecto cabe indicar, no obstante, los esfuerzos de modularización llevados a cabo por el Prof. Antonio Sarasa en su tesis doctoral, descritos en [Sarasa et al. 2008, Sarasa & Sierra 2013-b]. Efectivamente, en [Sarasa et al. 2008] se propone, como ya se ha comentado anteriormente, la especificación de las tareas de procesamiento mediante diferentes esquemas de traducción que se sincronizan entre sí a través de los valores semánticos computados. Por su parte, en [Sarasa & Sierra 2013-b] se propone la descomposición de tareas de procesamiento

complejas en subtareas más simples, y la especificación de cada subtarea mediante un fragmento de gramática de atributos. La gramática de atributos global surge, de esta forma, como resultado de unir todos estos fragmentos. No obstante, en dichas propuestas los distintos esquemas de traducción o los distintos fragmentos de gramáticas de atributos deben formularse sobre la misma gramática específica de procesamiento. Este hecho contradice, en cierta forma, el espíritu de uso de las gramáticas específicas para el procesamiento de ofrecer una estructura adecuada al tipo de procesamiento a llevar a cabo. Efectivamente, la estructura que puede resultar adecuada para una determinada subtarea o un determinado aspecto, puede no resultar adecuada para otros diferentes.

De esta forma, los siguientes dos capítulos están orientados a paliar estas dos limitaciones básicas de los enfoques dirigidos por lenguajes al procesamiento de documentos XML que hemos formulado en el grupo de investigación.

Capítulo 4
Formulación de Gramáticas Incontextuales Específicas para Procesamientos XML y Comprobación de su Conformidad respecto a Gramáticas Documentales

4.1 Introducción

Tal y como se ha visto en el capítulo anterior, en el proceso de desarrollo dirigido por lenguajes de aplicaciones de procesamiento XML adquiere una gran importancia la formulación de una gramática incontextual que defina una estructura sintáctica adecuada a la tarea de procesamiento que se quiere realizar. Para ello se toma, como punto de partida, la gramática documental (DTD u otro tipo de esquema documental) que define el lenguaje de marcado bajo consideración, y se proporciona una gramática incontextual específicamente orientada a la tarea de procesamiento a llevar a cabo. La gramática incontextual específica para el procesamiento resultante debe preservar, por tanto, el lenguaje generado por la gramática documental inicial. No obstante, la comprobación de esta restricción puede conducir, en último término, al problema de comprobar si dos gramáticas incontextuales son equivalentes entre sí, un problema que en, el caso general, es indecidible [Bar-Hillel et al. 1961]. Aunque restringiendo el nivel de ambigüedad de las gramáticas involucradas es posible convertir el problema en uno decidible [Baeten et al. 1993], las complejidades involucradas [Mayr 2000] no son prácticas para incluir el método en un sistema de desarrollo interactivo, en el que se precisan tiempos de respuesta muy rápidos.

De esta forma, en esta tesis se propone aplicar un método sistemático consistente en el modelizado gramatical de los modelos de contenidos de los tipos de elementos básicos del lenguaje. Dado que dichos modelos de contenidos definen lenguajes regulares, se podrá restringir el tipo de las gramáticas que realizan dichos modelos de contenidos a gramáticas que, en última instancia, definan también lenguajes regulares. Como consecuencia será posible organizar la gramática específica del procesamiento en una colección de subgramáticas que realizan modelos de contenidos, y que generan lenguajes regulares, y comprobar la equivalencia de dichas subgramáticas con respecto a los modelos de contenidos originales. Como resultado se obtiene un criterio de comprobación de la corrección de la gramática específica para el procesamiento con respecto a la gramática documental original más fuerte que el de la mera comprobación de la equivalencia, pero útil en la práctica. Cuando la aplicación de dicho criterio tenga éxito, diremos que la gramática específica para el

procesamiento es *conforme* con la gramática documental. De esta forma, el método de comprobación de la conformidad es un método intermedio entre la comprobación de la equivalencia *débil* de dos gramáticas incontextuales (comprobación de que las gramáticas generan el mismo lenguaje) y la equivalencia *estructural* (comprobación de que el conjunto de árboles de derivación generados por una gramática es igual, módulo una identificación apropiada de símbolos gramaticales, al conjunto de árboles generado por la otra). La comprobación de la equivalencia estructural sí es un problema decidible para gramáticas arbitrarias [Paull & Unger 1968] y resoluble de manera razonablemente eficiente en la práctica.

La estructura de este capítulo es como sigue. Primeramente, se describe en la sección 4.2, el proceso de formulación de gramáticas específicas para el procesamiento y la comprobación de la conformidad respecto a la gramática documental. La sección 4.3 revisa algunos elementos de la teoría de lenguajes formales que pueden emplearse para automatizar la comprobación de la conformidad. La sección 4.4 desarrolla un método para la conversión del tipo de gramáticas que realizan los modelos de contenidos en el enfoque propuesto (gramáticas *no autoembebibles*) en expresiones regulares. La sección 4.5 propone, entonces, un método de comprobación de la conformidad que presta especial atención al factor humano en el proceso de desarrollo. La sección 4.6 describe la evaluación del proceso general a fin de validar su utilidad práctica, dando lugar a una generalización del método inicialmente aportado, una herramienta para facilitar la realización de este proceso, y resultados de evaluación en un escenario real. Finalmente, en la sección 4.7, se presentan las conclusiones obtenidas.

En este capítulo se desarrolla, refina y amplía el trabajo desarrollado en [Temprado et al. 2011]. La propuesta para utilizar gramáticas no autoembebibles para realizar gramaticalmente modelos de contenidos XML se realiza en [Temprado et al. 2010-a].

4.2 Visión General del Enfoque

El proceso de formulación de una gramática específica para el procesamiento de un determinado tipo de documentos XML permite elaborar una gramática BNF con una estructura específicamente orientada al procesamiento de dichos documentos, e incluye soporte automático para la comprobación de la conformidad. Para conseguir dicho propósito, se propone el flujo de trabajo presentado en la Figura 4.2.1, cuyas actividades consisten en:

- Modelizado EBNF del lenguaje de marcado. Esta actividad consiste en la elaboración de una gramática EBNF que modeliza la estructura básica del lenguaje de marcado, ignorando aspectos complementarios tales como atributos u otra información no relevante de cara a la estructura. Las reglas EBNF de esta gramática modelizarán en sus cuerpos los modelos de contenidos (definidos mediante expresiones regulares) de cada elemento de la gramática documental bajo un símbolo no terminal que lo represente. Incluirán también símbolos terminales

representativos asociados con las etiquetas de apertura, cierre y contenido textual (#PCDATA) XML. En este proceso, si se considera oportuno podrán omitirse algunos elementos sustituyendo las ocurrencias de los correspondientes símbolos no terminales por sus correspondientes definiciones, así como introducir otros símbolos no terminales intermedios (asociados, por ejemplo, con las entidades tipo parámetro en una DTD). Como resultado, se obtiene una gramática EBNF a partir de la gramática documental, que abstrae la estructura básica de la misma, y que sirve como base a la realización de una gramática BNF con una estructura más adecuada al procesamiento. Los símbolos no terminales de la gramática EBNF que se obtiene en dicha actividad se denominarán *símbolos de núcleo*.

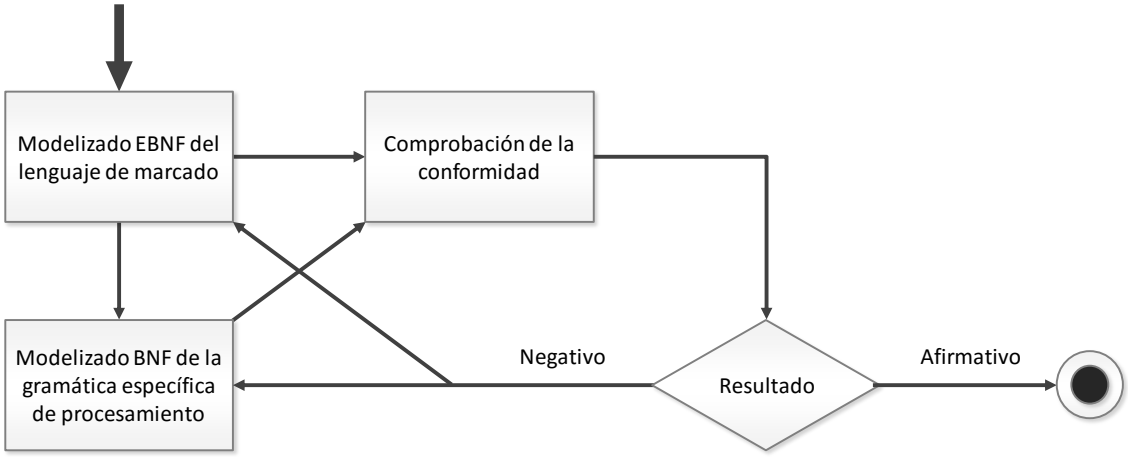


Figura 4.2.1. Proceso de formulación de gramáticas específicas para el procesamiento y comprobación automática de la conformidad.

- Modelizado BNF de la gramática específica de procesamiento. Esta gramática incontextual incluirá, entre otros no terminales, los símbolos de núcleo. Incluirá también los terminales de la gramática EBNF. Así mismo, realizará el cuerpo de cada producción asociada con cada símbolo de núcleo en la gramática EBNF mediante una subgramática *no autoembebible* [Langendoen 1975]. Una gramática no autoembebible es una gramática en la que no existen derivaciones de la forma $A=>^{*}\alpha A\beta$, con α, $\beta \neq \lambda$. La racionalidad para esta elección reside en que las gramáticas no autoembebibles son la clase de gramáticas BNF que menos limitaciones impone y mayor flexibilidad ofrece para describir lenguajes regulares. Por tanto, suponen una elección idónea para realizar gramaticalmente las expresiones regulares asociadas con los símbolos de núcleo.

- Comprobación de la conformidad. La comprobación de la conformidad se reduce, en el enfoque propuesto, a comprobar que cada subgramática asociada con cada símbolo de núcleo es equivalente (en el sentido de generar el mismo lenguaje) que la correspondiente expresión regular en la gramática EBNF que modeliza la estructura básica del lenguaje de marcado.

La respuesta negativa que el sistema puede aportar no finaliza el flujo de trabajo. Por el contrario, propugna revisar el estado de las gramáticas para descubrir posibles errores, o para ajustar dichas gramáticas mejor a las necesidades del problema a resolver. A este respecto, es importante observar que, dado que el modelizado EBNF del lenguaje de marcado puede haberse manipulado para eliminar no terminales con estructura trivial, o para introducir no terminales intermedios que expliciten estructuras relevantes implícitas en los modelos de contenidos, el proceso ofrece cierta flexibilidad a la hora de determinar los no terminales de núcleo más relevantes. De esta forma, no únicamente la gramática BNF específica para el procesamiento, sino también el modelizado EBNF del lenguaje de marcado, están sujetos a cambios. Este carácter dota al proceso de la propiedad de ser iterativo e incremental, potenciando la construcción y obtención de gramáticas de una calidad superior a la de las posiblemente concebidas originalmente.

La Figura 4.2.2 muestra un ejemplo para las dos primeras actividades del proceso sobre el lenguaje de las formas XML (ya utilizada a lo largo del Capítulo 2). Los símbolos no terminales de núcleo se muestran en **negrita**, los símbolos no terminales adicionales en la gramática BNF en gris, y los símbolos terminales en formato normal.

DTD

```
<?xml version="1.0" encoding="UTF-8" ?>

<!ENTITY % Min3Puntos '(Punto2D,Punto2D+,Punto2D)'>

<!ELEMENT Formas (Forma)*>
<!ELEMENT Forma (Nombre, Puntos)> <!ATTLIST Forma tipo CDATA #IMPLIED>
<!ELEMENT Nombre (#PCDATA)>
<!ELEMENT Puntos (%Min3Puntos;)>
<!ELEMENT Punto2D (CoordenadaX, CoordenadaY)>
<!ELEMENT CoordenadaX (#PCDATA)>
<!ELEMENT CoordenadaY (#PCDATA)>
```

Gramática EBNF

fs → <Formas> **f*** </Formas>
f → <Forma> **n ps** </Forma>
ps → **min3ps**
min3ps → **p2D p2D+ p2D**
p2D → <Punto2D> #PCDATA </Punto2D>
cx → <CoordenadaX> #PCDATA </CoordenadaX>
cy → <CoordenadaY> #PCDATA </CoordenadaY>
n → <Nombre> #PCDATA </Nombre>

Gramática BNF

fs ::= <Formas> fsr </Formas>
fsr **::= f** fsr
fsr **::=** λ
f ::= <Forma> **n ps** </Forma>
ps ::= min3ps
min3ps ::= pss
pss **::=** pst psr
psr **::= p2D** psr
psr **::=** λ
pst **::= p2D p2D p2D**
p2D ::= <Punto2D> #PCDATA </Punto2D>
cx ::= <CoordenadaX> #PCDATA </CoordenadaX>
cy ::= <CoordenadaY> #PCDATA </CoordenadaY>
n ::= <Nombre> #PCDATA </Nombre>

Figura 4.2.2. Ejemplo de gramática documental (DTD superior), modelizado de gramática EBNF (izquierda) a partir de la anterior, y modelizado de gramática BNF (derecha) a partir de la última.

Obsérvese, finalmente, que el problema crítico a resolver para que el modelo de proceso propuesto sea factible es encontrar formas adecuadas para realizar la actividad de comprobación de la conformidad. Este problema se reduce, en otros términos, a verificar que una gramática BNF es *conforme* a una gramática ENBF en el sentido siguiente : (i) la gramática BNF realiza todos y cada uno de los no terminales en la gramática EBNF (símbolos no terminales de núcleo), y (ii) la realización implica que las subgramáticas que definen los no terminales de núcleo (con respecto a otros símbolos no terminales de núcleo y símbolos terminales) son equivalentes a las correspondientes expresiones regulares en la gramática EBNF. La resolución de este problema se convertirá, por tanto, en el eje central del resto del capítulo.

4.3 Elementos para la comprobación de la conformidad

La comprobación de la conformidad puede llevarse a cabo: (i) transformando las expresiones regulares que definen los no terminales de núcleo en la gramática EBNF en autómatas finitos deterministas equivalentes, (ii) transformando las subgramáticas no autoembebibles que realizan dichas expresiones en la gramática BNF específica para el procesamiento en autómatas finitos deterministas equivalentes, y (iii) comprobando la equivalencia de los autómatas deterministas resultantes. Así mismo, en caso de que la comprobación resulte negativa, es conveniente que el proceso proporcione información útil al modelizador para el diagnóstico del problema.

De esta forma, en esta sección se revisan algunos elementos de la teoría de lenguajes formales que pueden resultar de utilidad de cara a soportar dicho proceso. Más concretamente, la sección 4.3.1 introduce algunos conceptos básicos relativos a expresiones regulares y autómatas finitos. La sección 4.3.2 revisa un algoritmo clásico para la comprobación de la equivalencia de dos autómatas finitos deterministas. La sección 4.3.3 revisa el método clásico de determinización de autómatas finitos no deterministas, al que se ha hecho ya alusión en el Capítulo 2. La sección 4.3.4 revisa las principales alternativas para traducir expresiones regulares en autómatas finitos. Por último, la sección 4.3.5 revisa el método más extendido para la transformación de gramáticas no autoembebibles en autómatas finitos.

4.3.1 Expresiones regulares y autómatas finitos

Las expresiones regulares son un formalismo declarativo para la descripción de lenguajes regulares [Hopcroft & Ullman 1979] (en lo que sigue, dichas expresiones se denotarán mediante letras griegas minúsculas α, β, etc.). El formalismo permite describir patrones de cadenas mediante operaciones básicas de concatenación, suma e iteración, que se aplican partiendo de símbolos de un alfabeto. De esta forma:

- Como expresiones regulares básicas se consideran: (i) $\varnothing$, que denota el lenguaje vacío, (ii) λ, que denota el lenguaje formado por la cadena vacía, y (iii) expresiones de la forma *a*, con *a* un símbolo del alfabeto (una expresión *a* de dicho tipo denota el lenguaje formado por una única cadena formada por dicho símbolo: "a").

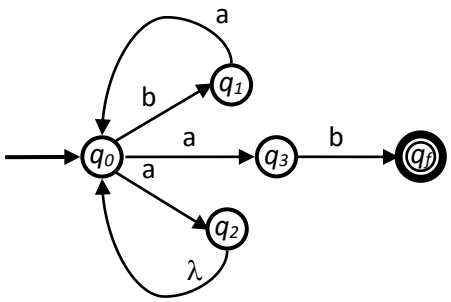
- Por su parte, las expresiones regulares más simples α y β pueden combinarse para dar lugar a expresiones regulares más complejas. De esta forma, la expresión *suma* α+β denota la unión de los lenguajes denotados por α y β (también es usual utilizar | como operador para la suma). La expresión *concatenación* α·β (o, simplemente, αβ) denota la concatenación de dichos lenguajes. Por último, la expresión *iteración* o *cierre de Kleene* α* denota el lenguaje formado por cero ({λ}), una o más concatenaciones sucesivas del lenguaje denotado por α. De esta manera, la suma y concatenación se conciben como la aplicación de operaciones binarias y, la iteración como la aplicación de una operación unaria, sobre expresiones regulares.

Por ejemplo, la expresión regular (ab+cb), es una expresión regular compuesta por la expresión regular α = ab y β = cb como α+β, cuya suma define el lenguaje regular formado por la cadena "ab" y la cadena "cb". De esta forma, pueden describirse expresiones regulares sintácticamente diferentes, pero semánticamente idénticas. Por ejemplo, (ab+cb) genera el mismo lenguaje que (cb+ab), y por lo tanto ambas son *equivalentes*. En consecuencia, las expresiones regulares pueden simplificarse sintácticamente mediante identidades como $\alpha+\varnothing=\alpha$, $\varnothing+\alpha=\alpha$, $\varnothing\alpha=\varnothing$, $\alpha\varnothing=\varnothing$, $\lambda\alpha=\alpha$, $\alpha\lambda=\alpha$, $\alpha+\alpha=\alpha$, $\alpha+(\beta+\gamma)=(\alpha+\beta)+\gamma$, $\varnothing^*=\lambda$, $\lambda^*=\lambda$, $(\alpha^*)^*=\alpha^*$, etc. En particular, dos expresiones regulares que pueden ser reducidas a la misma expresión por aplicación de la propiedad conmutativa ($\alpha+\beta=\beta+\alpha$), asociativa ($\alpha+(\beta+\gamma)=(\alpha+\beta)+\gamma$) e idempotente ($\alpha+\alpha=\alpha$) de la suma de expresiones regulares se denominan ACI-*similares* [Brzozowski 1964].

Por su parte, los autómatas finitos [Hopcroft & Ullman 1979] son un formalismo operacional para la descripción de modelos reconocedores que aceptan las cadenas de un lenguaje regular. Un autómata finito está formado por estados (que se denotarán mediante *q*, *q'*, *Q, Q',* etc.) y transiciones entre estados a través de un símbolo del alfabeto *a* (que se denotarán como *q-a->q'*, evidenciando que *a* es la etiqueta del enlace dirigido desde el estado *q al estado q'*). Uno de los estados se distingue como estado inicial (comúnmente denotado con el subíndice cero, como q_0), a partir del cual comienza el reconocimiento. También pueden existir estados finales, en los cuáles termina dicho reconocimiento. Las transiciones pueden ser provocadas por la lectura del siguiente símbolo *a* del resto de la sentencia de la entrada (*q-a->q'*), o bien pueden ser esporádicas, sin requerir reconocer símbolo alguno (λ-transiciones, *q*-λ-> *q'*). El autómata es *determinista* (AFD) si no contiene λ-transiciones y para cada símbolo *a* y estado *Q*, existe como mucho una única transición *Q-a->Q*'. En otro caso el autómata poseerá carácter *no determinista* (AFND). En lo sucesivo, se utilizarán letras mayúsculas para referirse a los estados de un autómata determinista, y letras minúsculas para referirse a los estados de uno no determinista. Así mismo, como ya se ha hecho patente en el

Capítulo 2, los autómatas finitos pueden describirse gráficamente mediante diagramas de transiciones, en los que los estados son nodos y las transiciones arcos.

AFND para la expresión regular (ba+a)*ab



AFD para la expresión regular (ba+a)*ab

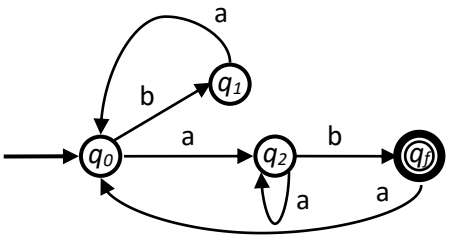


Figura 4.3.1. Ejemplo de un AFND y un AFD equivalentes a una expresión regular.

Como ejemplo ilustrativo, se expone en la Figura 4.3.1, para la expresión regular α=(ba+a)*ab, un autómata finito no determinista reconocedor equivalente, y otro determinista también equivalente (los estados finales se representan como un círculo con doble borde).

4.3.2 Comprobación de la equivalencia de dos autómatas finitos deterministas

Cuando dos autómatas finitos deterministas reconocen todas y cada una de las sentencias pertenecientes a un lenguaje, y únicamente dichas sentencias, entonces ambos autómatas se caracterizan por ser *equivalentes* entre sí. Los métodos existentes para llevar a cabo tal comprobación se basan, en mayor o menor medida, en el algoritmo propuesto en [Hopcroft & Karp 1971], tal y como se reconoce en [Almeida et al. 2010]. El funcionamiento del método se basa en las tres reglas siguientes, que deben cumplir todo par de AFDs equivalentes entre sí:

- Los estados iniciales Q_0 y Q'_0 de ambos autómatas deben ser *equivalentes*. Formalmente, $Q_0 \sim Q'_0$.
- Si dos estados han de ser *equivalentes* entre sí, entonces cada transición por cada símbolo del alfabeto debe conducir a un nuevo par de estados equivalentes entre sí. Formalmente, si Q~Q', Q-*a*->*P*', Q'-*a*->*P*', entonces *P*~*P*'.
- Ningún estado final puede ser equivalente a un estado no final.

Entrada: AFD y AFD', cuya equivalencia o no equivalencia quiere determinarse.
Salida: Juicio sobre si AFD y AFD' son equivalentes/ no son equivalentes entre sí.
Método:
- Para cada estado *Q* de AFD y de AFD', determinar la clase de equivalencia de *Q* como {*Q*}.
- Sean Q_0 y Q'_0 los estados iniciales de AFD y AFD' respectivamente:
  - Si Q_0 es final y Q'_0 no lo es, o viceversa, **terminar:** decidir que los autómatas ***no son equivalentes***.
  - Marcar el objetivo $Q_0 \sim Q'_0$ como a explorar y fijar la clase de equivalencia de Q_0 y de Q'_0 a $\{Q_0, Q'_0\}$.
- Mientras haya objetivos a explorar**:**
  - Elegir uno de estos objetivos $Q \sim Q'$. Desmarcarlo como objetivo a explorar
  - Para cada símbolo *a*:
    - Considerar las transiciones $Q \text{ –}a\text{-> } P$ y $Q' \text{ –}a\text{-> } P'$.
    - Si *P'* es final y *P'* no lo es, o viceversa, **terminar**: **decidir** que los autómatas ***no son equivalentes***.
    - Si la clase de equivalencia de *P* es distinta de la clase de equivalencia de *P'*
      - Marcar el *objetivo* $P \sim P'$ como objetivo a explorar.
      - Sea Γ la clase de equivalencia de *P* y Γ' la clase de equivalencia de *P'*. Para cada estado *R* en $\Gamma \cup \Gamma'$ fijar la clase de equivalencia de *R* a $\Gamma \cup \Gamma'$.
- **Decidir:** los autómatas ***son equivalentes***.

Figura 4.3.2. Algoritmo de comprobación de equivalencia entre dos autómatas finitos deterministas.

De esta forma, el algoritmo adopta una estrategia de refutación basada en dichas reglas. Para ello, parte del aserto inicial $Q_0 \sim Q'_0$, tratando de refutar, en cada paso, un aserto de la forma Q~Q' aún no explorado. Para ello, trata de encontrar un símbolo para el cuál se transite desde *Q* a un estado no final y desde *Q*' a uno final, o viceversa. En caso de no tener éxito, añade los nuevos asertos *P*~*P*' para cada símbolo *a* y para cada par de transiciones Q-*a*->*P* y Q'-*a*->*P'* como asertos a verificar, *a no ser que P* y *P'* tengan asignada la misma *clase de equivalencia*. Efectivamente, para podar el proceso de refutación, el algoritmo agrupa los estados en *clases de equivalencia*, asignando a cada estado el conjunto de estados que deben ser equivalentes al mismo. De esta forma, si se plantea probar la equivalencia de dos estados que están incluidos en la misma clase de equivalencia, el aserto se resuelve sin más. Como consecuencia, el algoritmo puede terminar agotando todas las posibilidades (en este caso, los autómatas serán equivalentes), o bien descubriendo dos estados que no son equivalentes. El algoritmo se muestra en la Figura 4.3.2.

Es interesante destacar que es sencillo extender el algoritmo para computar, en caso de no equivalencia, una cadena que conduzca a estados no equivalentes. Por otra parte, el método presenta una complejidad reducida en tiempo de ejecución, acotada por el producto del número de estados del autómata más grande y el tamaño del alfabeto del lenguaje.

4.3.3 Determinización de autómatas finitos no deterministas

La mayoría de las veces, las expresiones regulares o las gramáticas no autoembebibles se convierten a autómatas finitos no deterministas equivalentes como paso previo a su determinización. De esta forma, es necesario realizar la conversión de un autómata finito no determinista a uno determinista equivalente. Para realizar dicha transformación, basta con aplicar el método de construcción por subconjuntos [Aho et al. 2006] al que ya se hizo referencia en el Capítulo 2. El método se fundamenta en obtener un nuevo autómata determinista cuyos estados son conjuntos de estados del autómata no determinista original. Para ello se aplica sobre el AFND el algoritmo que se muestra en la Figura 4.3.4.

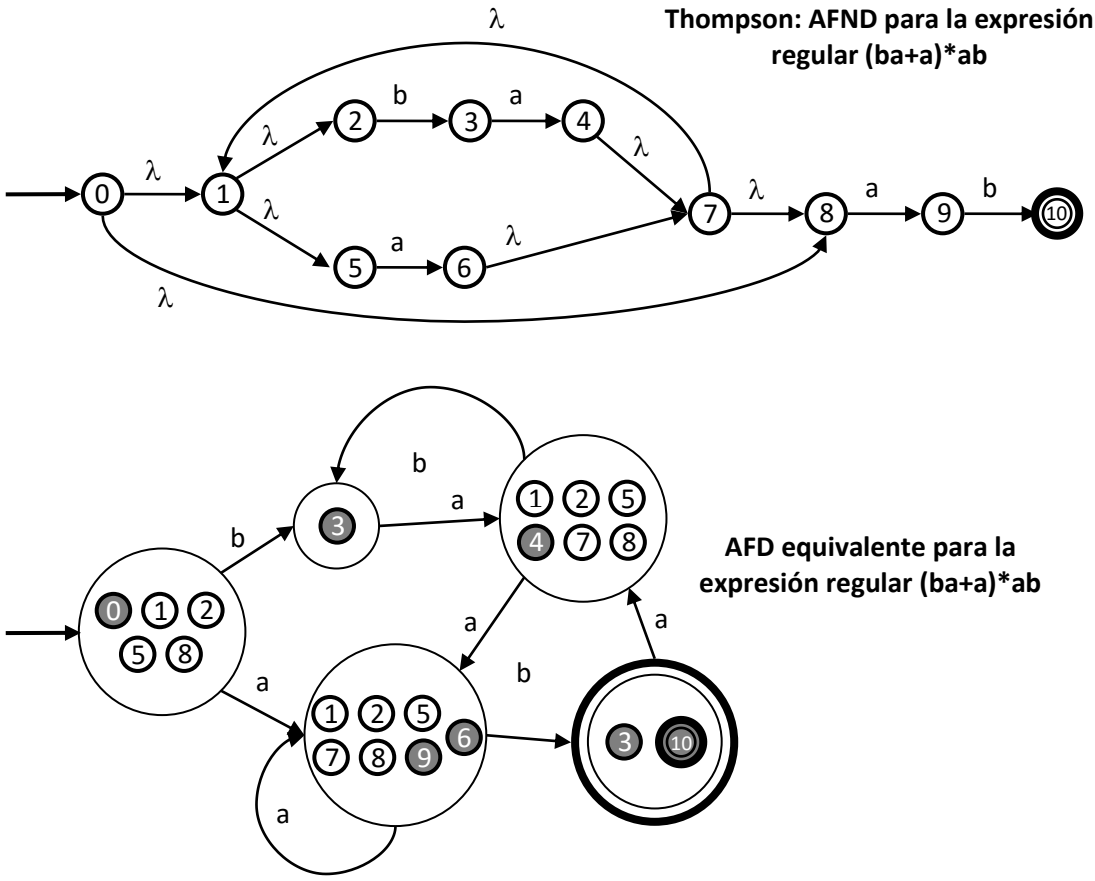


Figura 4.3.3. AFND (Thompson) y su AFD equivalente por el método de construcción por subconjuntos.

La Figura 4.3.3 muestra el AFND que reconoce la expresión regular (ba+a)*ab y su transformación equivalente a un AFD por el método de construcción por subconjuntos. En la figura, los estados coloreados en blanco se calculan por aplicación de la operación de λ-*cierre* a partir de los estados grises. Así mismo, nótese que, en el caso de AFNDs sin λ-transiciones, es posible evitar la operación de λ-cierre, lo que simplifica el algoritmo.

Entrada: AFND a determinizar.
Salida: AFD equivalente
Método:
- A partir del estado inicial q_0 del AFND se construye el estado inicial Q_0 del AFD como el λ-*cierre* de $\{q_0\}$. Q_0 se marca como estado a explorar (el λ-cierre de un conjunto de estados Q se calcula iterativamente como la mínima solución de λ-*cierre de* $Q = Q \cup \{q' \mid q \in \lambda\text{-cierre de } Q \text{ y } q\text{-}\lambda\text{->}q'\}$).
- Mientras haya estados a explorar:
    - Elegir un estado a explorar Q y desmarcarlo como estado a explorar.
    - Para cada símbolo a:
        - Determinar Q' como el λ-cierre de $\{q' \mid q \in Q \text{ y } q\text{-}a\text{->}q'\}$.
        - Si Q' no está en el AFD, añadirlo y marcarlo como estado a explorar.
        - Añadir la transición $Q\text{-}a\text{->}Q'$.

Figura 4.3.4. Algoritmo de determinización de autómatas por el método de los subconjuntos.

4.3.4 Transformación de expresiones regulares en autómatas finitos

En esta sección se presentan los diferentes métodos de transformación de expresiones regulares en autómatas finitos reconocedores. Cada uno de estos métodos se plantea como una técnica diferente para hallar un autómata finito que reconoce las mismas sentencias del lenguaje que la expresión regular a transformar. Debido a la manera característica que posee cada método para hallar este autómata, estos métodos se diferencian de los demás por el tipo de autómata (AFD o AFND como ocurre en la mayoría de los casos) y número de estados y transiciones que poseerá éste como resultado. En caso de obtener un AFND siempre podrá hallarse un AFD equivalente mediante la aplicación del método de construcción por subconjuntos de la sección 4.3.3. A continuación, se detallarán cada uno de los métodos principales.

4.3.4.1 Algoritmo de Thompson

En [Thompson 1968] se plantea un método directo e intuitivo a la hora de transformar una expresión regular en un autómata finito no determinista equivalente. La construcción sigue un enfoque de traducción dirigida por la sintaxis, produce autómatas con un único estado final, y se realiza atendiendo a transformaciones directas y recursivas de casos sencillos en función de si una expresión regular es:

- El conjunto vacío $\varnothing$. Produce un único estado, no final.

- La cadena vacía λ. Produce un estado inicial que transita a un estado final mediante una λ-transición.

- Un símbolo del alfabeto *a*. Produce un estado inicial que transita a un estado final por dicho símbolo *a*.

- Una suma de expresiones regulares α+β. Obtenidos los autómatas para α y β, produce un estado inicial con una λ-transición al estado inicial del autómata para α y una λ-transición al estado inicial del autómata para β. Se crea un estado final, y se añaden λ-transiciones a dicho estado desde los estados finales de los autómatas para α y β, estados que se convierten en no finales.

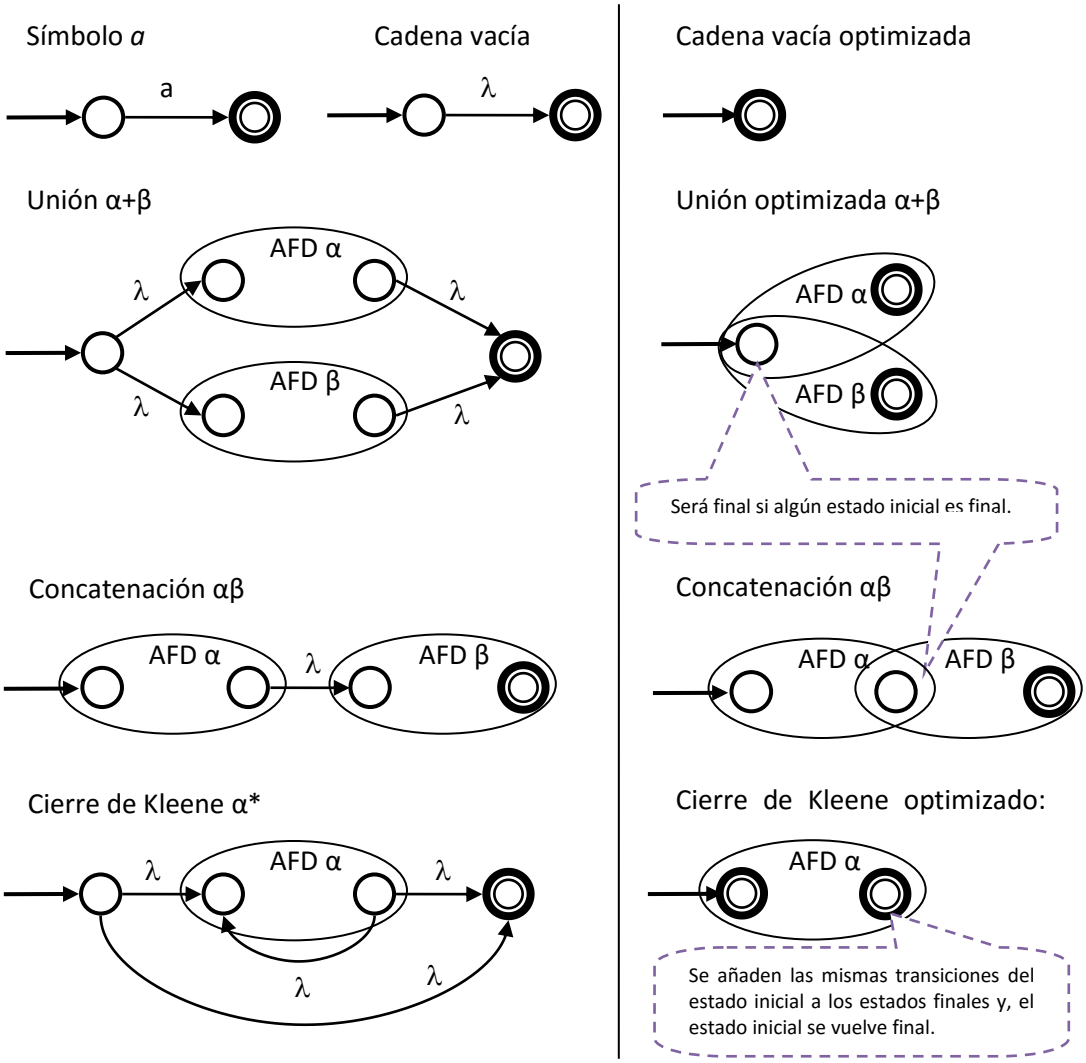


Figura 4.3.5. Construcción de Thompson de un AFND a partir de una expresión regular (izquierda) y una optimización esquemática que introduce menos λ-transiciones (derecha).

- Una concatenación de expresiones regulares αβ. Obtenidos los autómatas para α y β, se considera estado inicial al estado inicial del autómata α, se añade una λ-transición desde el estado final de dicho autómata al estado inicial del autómata para β, y dicho estado final se convierte en no final.

- Un cierre de Kleene α*. Obtenido el autómata para α, produce un estado inicial con una λ-transición al estado inicial del autómata α, una λ-transición desde el estado final del autómata α (que se convierte en no final) a su estado inicial y otra λ-transición a un nuevo estado final que es creado. También produce una λ-transición del estado inicial anteriormente creado a dicho nuevo estado final.

Esta construcción del autómata puede mejorarse en el número de estados y transiciones que se crean en varios de los casos. La creación de transiciones vacías o λ-transiciones no es deseable, pues complica las posteriores determinizaciones. La Figura 4.3.5 ilustra la construcción recursiva de expresiones regulares en autómatas finitos no deterministas mediante el algoritmo de Thompson, y cómo realizar una optimización que introduce menos λ-transiciones (véase [Ilie & Yu 2003] para una propuesta de optimización que permite la posterior eliminación de las λ-transiciones en tiempo cuadrático con relación al tamaño de la expresión regular). Adicionalmente, la Figura 4.3.3 presentó una transformación de una expresión regular mediante este método, donde se empleó el caso optimizado para la concatenación.

4.3.4.2 Algoritmo de las derivadas

En [Brzozowski 1964] se propone un método para la construcción directa de un autómata finito determinista a partir de una expresión regular. Dicho método se basa en el concepto de *derivada* de una expresión regular. Dada una expresión regular α y un símbolo *a*, la derivada de α con respecto a *a* se denota por δ(α, *a*) y se define mediante las siguientes reglas:

- Casos básicos:

  δ(∅, *a*) = ∅.
  δ(λ, *a*) = ∅.
  δ(*a*, *a*) = λ.
  δ(*b*, *a*) = ∅ si *b* ≠ *a*.

- Casos recursivos.

  δ(α+β, *a*) = δ(α, *a*) + δ(β, *a*).
  δ(αβ, *a*) = δ(α, *a*)β+ε, donde:
  Si α es *anulable* (es decir, puede producir la cadena vacía λ) ε = δ(β, *a*), en otro caso ε = ∅.
  δ(α*, *a*) = δ(α, *a*)α*.

La construcción del autómata es de la siguiente forma:

- Inicialmente se crea el estado inicial del autómata, que se corresponde con la expresión regular original.

- Para cada estado creado Q asociado con α y cada símbolo *a* del alfabeto, se calcula la derivada $\delta(\alpha, a)$. Si no se ha creado ya un estado Q' asociado con β, siendo β *equivalente* a $\delta(\alpha, a)$, entonces se crea tal estado Q' y se asocia con $\delta(\alpha, a)$. Se añade, por último, la transición Q –*a*-> Q'.

- El estado final es aquel cuya expresión derivada es equivalente a la cadena vacía λ.

En [Brzozowski 1964] se demuestra la corrección de este método, así como el carácter mínimo del autómata resultante. No obstante, el principal inconveniente del mismo es que se basa en la determinación de la equivalencia entre expresiones regulares (este es, de hecho, el problema que queremos resolver con ayuda de los métodos que estamos presentando). Afortunadamente, como se muestra también en [Brzozowski 1964], la equivalencia se puede sustituir por la condición más débil de *congruencia* módulo ACI-similitud (es decir, mediante la normalización de las expresiones por aplicación de las reglas de asociatividad, conmutatividad e idempotencia de la suma). Así mismo, también pueden aplicarse otras reglas de simplificación de expresiones regulares para producir autómatas más pequeños. La Figura 4.3.6 muestra un ejemplo del autómata finito determinista producido por la expresión regular (ba+a)*ab mediante el algoritmo de derivadas.

Derivadas: AFD para la expresión regular (ba+a)*ab

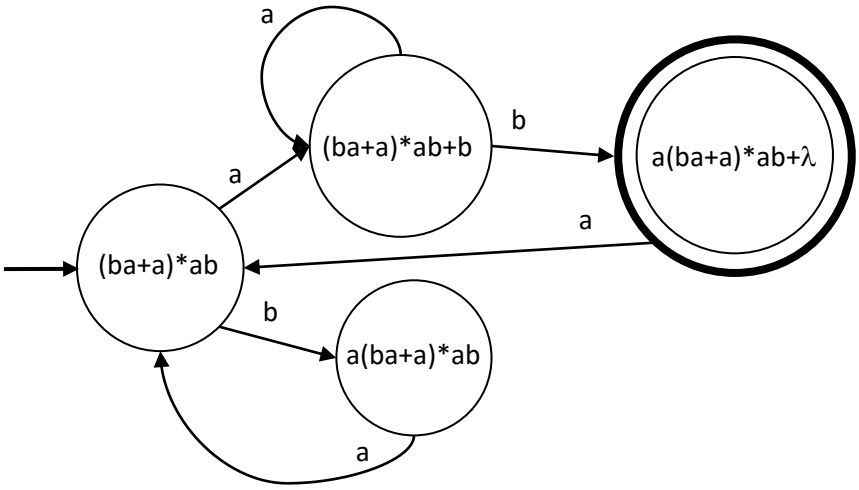


Figura 4.3.6. AFD construido a partir de una expresión regular por el método de derivadas.

El principal atractivo de este método es proporcionar un método directo para convertir una expresión regular en un autómata finito determinista equivalente. Desafortunadamente, para ciertos casos, el número de estados que produce, aún con el uso de reglas de simplificación, puede ser muy elevado, y la complejidad de las derivadas muy grande.

4.3.4.3 Algoritmo de Berry-Sethi

El algoritmo descrito en [Berry & Sethi 1986] construye un autómata finito no determinista a partir de una expresión regular fundamentándose en el principio de derivación de expresiones regulares [Brzozowski 1964]. La propuesta surge como una forma más eficiente a este algoritmo de construcción del autómata mediante derivadas introduciendo marcado de posiciones para las mismas. Aunque produce autómatas no deterministas, su complejidad es reducida respecto al algoritmo de derivadas. Para ello, cada símbolo presente en la expresión regular original queda identificado mediante una posición única y que lo distingue de los demás símbolos. Esto se traduce en que, al aplicar la derivada por una posición *p* en una expresión regular α, las reglas de derivación se simplifican, ya que sólo existe una ocurrencia de *p* en α. De esta manera, en una expresión como (ba+a)*ab, al marcarse sus símbolos, por ejemplo, con números, quedan diferenciados entre sí, dando lugar $(b_1a_2+a_3)^*a_4b_5$, y produciéndose, de esta forma, un alfabeto de símbolos más amplio sobre los que derivar mediante las reglas de derivación de la sección 0. Con el marcado de posiciones de la expresión regular, cada símbolo de entrada de dicha expresión dará lugar a una única derivada (módulo equivalencia), obteniéndose un número de estados equivalente al número de símbolos del nuevo alfabeto marcado más el estado inicial. En relación con la implementación, el algoritmo puede realizarse de manera más óptima analizando los símbolos *primeros* y *sucesores* de las posiciones en las expresiones obtenidas. Concretamente se añade un símbolo adicional de terminación a la expresión regular marcada mediante "!". El proceso es el siguiente:

- Se crea un estado inicial y un estado por cada símbolo marcado de la expresión regular marcada. Estos estados se identifican por el símbolo marcado.

- Del estado inicial se genera una transición al estado del símbolo marcado ya creado si tal símbolo pertenece a los símbolos *primeros* de la expresión regular. Dicha transición estará etiquetada por el símbolo sin marcar.

- Para los restantes estados asociados a símbolos marcados, se crea una transición desde un estado a otro si el símbolo del estado destino pertenece a los símbolos *sucesores* del símbolo del estado origen. Dicha transición estará etiquetada por el símbolo sin marcar.

Los *símbolos primeros* de una expresión regular son por los que comienzan las cadenas del lenguaje denotado por la misma, mientras que los *símbolos sucesores* de un símbolo *a* son aquellos que pueden seguir a *a* en dichas cadenas. Véase en la Figura 4.3.7 el autómata resultante para la expresión regular (ba+a)*ab construido por el método de Berry-Sethi.

Berry-Sethi: AFND para la expresión regular (ba+a)*ab

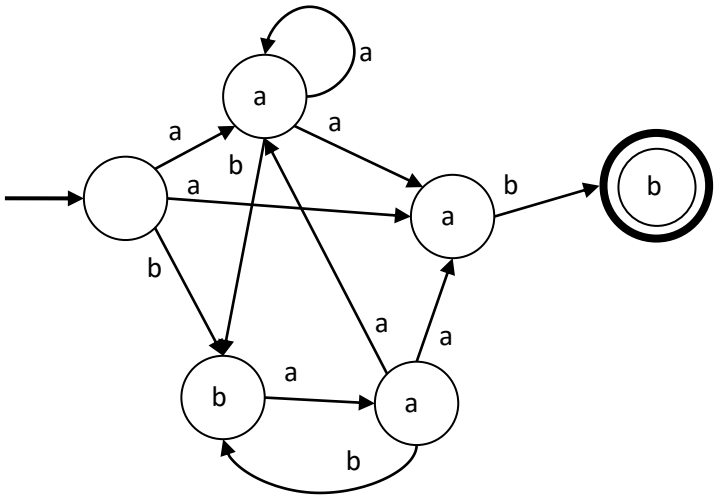


Figura 4.3.7. AFND construido a partir de una expresión regular por el método de Berry-Sethi.

Este método produce, por tanto, un autómata no determinista equivalente a la expresión regular de partida que, al contrario que el producido por el método de Thompson, no contiene λ-transiciones.

4.3.4.4 Algoritmo de derivadas parciales

El algoritmo de derivadas parciales presentado en [Antimirov 1996] y optimizado en [Champarnaud & Ziadi 2002] plantea la construcción de autómatas finitos no deterministas pequeños en número de estados, a partir de una expresión regular basándose en el algoritmo de [Brzozowski 1964]. El concepto de derivada parcial surge de normalizar las derivadas convencionales aplicando la propiedad distributiva de la concatenación con respecto a la unión (es decir, $\alpha(\beta+\gamma)=\alpha\beta+\alpha\gamma$ y $(\beta+\alpha)\gamma=\beta\gamma+\alpha\gamma$). Con ello, las derivadas pueden expresarse como sumas de la forma $\alpha_0+\alpha_1+\ldots$, donde cada α_i es una concatenación de expresiones básicas ($\varnothing$, λ, *a* …), o cierres de Kleene. Cada término α_i se denomina *derivada parcial*. Dichas derivadas parciales pueden calcularse directamente como un conjunto de expresiones durante el proceso de derivación, en lugar de realizar la normalización anteriormente aludida. Más concretamente, el conjunto de derivadas parciales $\partial(\alpha, a)$ para una expresión α respecto a un símbolo *a* queda determinado por las siguientes reglas, adaptación directa de las reglas de derivación de expresiones regulares:

- Casos básicos:

$\partial(\varnothing, a) = \varnothing$.
$\partial(\lambda, a) = \varnothing$.
$\partial(a, a) = \{\lambda\}$.
$\partial(b, a) = \varnothing$ si $b \neq a$.

- Casos recursivos.

$\partial(\alpha+\beta, a) = \partial(\alpha, a) \cup \partial(\beta, a)$.
$\partial(\alpha\beta, a) = \partial(\alpha, a)\beta \cup \varepsilon$. Si α es *anulable* $\varepsilon = \partial(\beta, a)$, en otro caso $\varepsilon = \varnothing$.
$\partial(\alpha^*, a) = \partial(\alpha, a)\alpha^*$.

donde la operación *concatenación* a un conjunto de expresiones se define como $\{\alpha_0+\ldots+\alpha_n\}\gamma = \{\alpha_0\gamma, \ldots \alpha_n\gamma\}$. Esta pequeña consideración reduce el número de derivadas generadas al no distinguir explícitamente expresiones regulares sintácticamente diferentes, pero semánticamente idénticas, tales como ab+c y c+ab, representándose ambas expresiones mediante un solo conjunto de expresiones regulares {ab, c}. De esta manera, cada expresión del conjunto es una derivada parcial.

Derivadas parciales: AFND para la expresión regular (ba+a)*ab

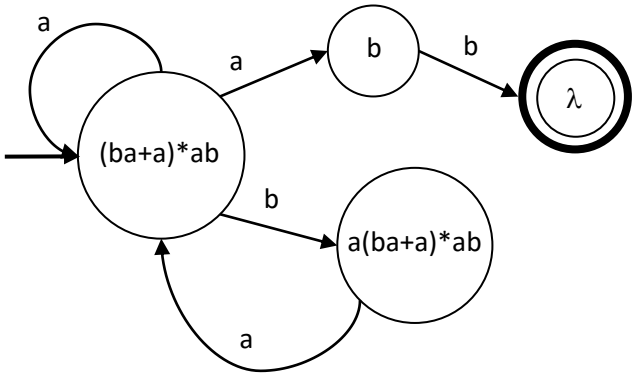


Figura 4.3.8. AFND construido a partir de una expresión regular por el método de derivadas parciales.

Siguiendo estas consideraciones, la construcción del autómata queda de la siguiente forma:

- El primer estado del autómata se asocia a la expresión regular original.

- Dado un estado *q* asociado con α, para cada símbolo *a* del alfabeto y cada β en $\partial(\alpha, a)$, se añade un estado *q*' asociado con β (si no existe ya) y una transición *q* –*a*-> *q*'. De esta forma, cada derivada parcial se corresponderá con un estado, y cada uno de sus estados destinos se corresponderán con cada una de las derivadas parciales que resultan de la derivación por un símbolo del alfabeto. De esta forma, a diferencia del algoritmo de [Brzozowski 1964], un estado con derivada parcial ab+ac transita a dos

estados destino al derivarse por *a*: b y c, en lugar de al único estado con derivada b+c, que sería generado por el algoritmo de derivadas y que, consecuentemente, daría lugar a nuevas expresiones derivadas.

- Los estados finales son aquellos cuya expresión derivada es la cadena vacía λ.

La Figura 4.3.8 muestra un ejemplo del autómata finito no determinista producido por el algoritmo de las derivadas parciales a partir de la expresión regular (ba+a)*ab.

Al igual que el método de Berry-Sethi, este método produce autómatas sin λ-transiciones.

4.3.5 Transformación de gramáticas no autoembebibles en autómatas finitos: algoritmo de Nederhof

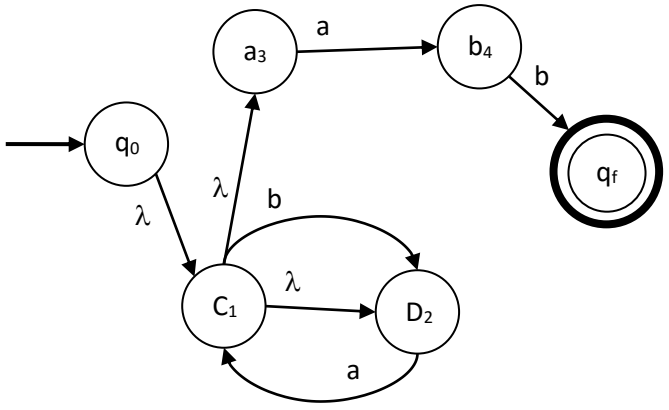
El último aspecto clave a resolver para la prueba de la conformidad es la transformación de gramáticas no autoembebibles en autómatas finitos. A este respecto, el algoritmo presentado en [Nederhof 2000] es un método eficiente y directo para obtener un autómata finito no determinista (con λ-transiciones) a partir de una gramática incontextual no autoembebible. El algoritmo comienza determinando la colección N_L de los no terminales *mutualmente recursivos a izquierdas* y la colección N_R de los no terminales *mutuamente recursivos a derechas*. Cada elemento de N_L es un conjunto Γ_L de no terminales de la gramática, en el que, seleccionado un par de no terminales A, B $\in \Gamma_L$ se cumple A $\Rightarrow^*$ Bα y B $\Rightarrow^*$ Aβ. Por su parte, cada elemento Γ_R de N_R es tal que, para cada A, B $\in \Gamma_R$ se cumple A $\Rightarrow^*$ αB y B $\Rightarrow^*$ βA. Obsérvese que la determinación de estas colecciones ofrece una condición suficiente para determinar si la gramática es no autoembebible: que cada no terminal aparezca en, a lo sumo, una de las dos colecciones. Efectivamente, si hay un no terminal A que aparece en algún conjunto de ambas colecciones, la gramática corre el riesgo de ser autoembebible, ya que puede darse la derivación A $\Rightarrow^*$ Aα por estar A en un conjunto de N_L, y la derivación A $\Rightarrow^*$ β*A* por estar A en un conjunto de N_R, resultando derivaciones de la forma A $\Rightarrow^*$ βAα. La prueba no es concluyente, ya que pudiera ocurrir que, en todos los casos, α, β = λ. No obstante, los métodos de determinación de N_L y N_R pueden extenderse fácilmente para discriminar no terminales para los que no es posible tal contingencia. De hecho, el algoritmo funciona correctamente para situaciones originadas por símbolos cíclicos (A $\Rightarrow^*$ B y B $\Rightarrow^*$ A), incorporando estos símbolos exclusivamente a una de las colecciones (N_R).

Una vez determinadas las colecciones N_L y N_R, y comprobado que la gramática es, efectivamente, no autoembebible, la obtención del AFND equivalente se realiza siguiendo un método de construcción recursivo en función de: (i) un estado origen q_o, (ii) una forma sentencial de guía, (iii) un estado destino q_d. El autómata construido (para la forma sentencial de guía) debe ser capaz de reconocer el lenguaje generado por la forma sentencial, partiendo del estado origen proporcionado, y terminando en el estado destino. Inicialmente se crea un estado inicial y un estado final, y la construcción recursiva se inicia con los mismos utilizando

como forma sentencial de guía el axioma de la gramática. En función de la forma sentencial de guía se aplican los siguientes casos:

- Es λ. Se crea una transición q_o -λ-> q_d.
- Es un terminal *a*. Se crea una transición q_o -*a*-> q_d.
- Un no terminal A que pertenece a un conjunto Γ en N_L. Para cada no terminal B de Γ se crea un nuevo estado q_B. Para cada regla B ::= σ (con σ resto de símbolos del cuerpo de la producción) en la que σ no comienza por un no terminal en Γ, se aplica la construcción para (i) q_o, (ii) σ, (iii) q_B. Para cada regla B ::= C σ, con C en Γ, se aplica la construcción para (i) q_C, (ii) σ, (iii) q_B. Por último, se incluye una λ-transición q_A -λ-> q_d.
- Un no terminal A que pertenece a un conjunto Γ en N_R. El tratamiento es simétrico al anterior. Aparte de crear un nuevo estado q_B para cada B en Γ, para cada regla B ::= σ en la que σ no termina en un no terminal en Γ, se aplica la construcción para (i) q_B, (ii) σ y (iii) q_d. Para cada regla B ::= σ C, con C en Γ, se aplica la construcción para (i) q_B, (ii) σ, (iii) q_C. Por último, se añade la transición q_o -λ-> q_A.
- Un no terminal no recursivo (no figura ni en N_L ni en N_R). Para cada regla A ::= β se aplica la construcción para (i) q_o, (ii) β, (iii) q_d.

Nederhof: AFND para la expresión regular (ba+a)*ab generada por la gramática:



S ::= C a b
C ::= b D | D | λ
D ::= a C

Figura 4.3.9. AFND construido a partir de una gramática no autoembebible por el método de Nederhof.

- Un símbolo (terminal o no terminal) *X* seguido de una cadena no vacía β, *X*β. Se crea un estado intermedio q_i, se aplica la construcción para (i) q_o, (ii) *X*, (iii) q_i, y se aplica la construcción para (i) q_i, (ii) β, (iii) q_d.

La Figura 4.3.9 ilustra el autómata no determinista resultante producido para una gramática no autoembebible que genera el lenguaje (ba+a)*ab y que presenta un conjunto recursivo a derechas formado por los no terminales C y D.

El cálculo de los conjuntos recursivos a izquierdas y derechas puede ser costoso. No obstante, como se propone en [Dilkes & Visnevski 2004], existe una manera eficiente de calcular estos conjuntos a partir de grafos que representan las relaciones de derivación inmediatas a izquierdas (derechas) entre no terminales. Efectivamente, los conjuntos buscados serán los componentes fuertemente conexos de dichos grafos, componentes que pueden determinarse eficientemente mediante el algoritmo de Tarjan [Tarjan 1972], lo que nos permite conocer las colecciones N_L y N_R y sus conjuntos Γ con un coste lineal en función del número de símbolos no terminales de la gramática. La principal característica del algoritmo de Nederhof es, con esta optimización añadida, el tratar, de manera muy eficiente, casos complejos en los que existen muchos no terminales mutualmente recursivos a izquierdas (derechas). No obstante, estos casos se presentan raramente cuando las gramáticas no autoembebibles se utilizan como mecanismos de modelizado gramatical de expresiones regulares.

4.4 Transformación de gramáticas no autoembebibles en expresiones regulares

Una alternativa a la aplicación del método de Nederhof (sección 4.3.5) a una gramática no autoembebible para obtener un AFND equivalente, consiste en aplicar un algoritmo de conversión de gramáticas no autoembebibles en expresiones regulares, y posteriormente, aplicar uno de los algoritmos de transformación de expresiones regulares en autómatas (AFND o AFD directo, sección 4.3.4). En esta tesis se ha explorado esta posibilidad. La racionalidad es obtener autómatas que permitan ofrecer mejores diagnósticos durante la prueba de conformidad, en caso de que el resultado de dicha prueba sea negativo. De esta manera, se ha diseñado en la sección 4.4.1 un método directo de transformación de gramáticas no autoembebibles en expresiones regulares, y una optimización de dicho método en la sección 4.4.2 para garantizar un óptimo rendimiento.

4.4.1 Método directo

A continuación, se detalla el funcionamiento del método directo de transformación de una gramática no autoembebible en una expresión regular. Este método se basa en el presentado en [Andrei et al. 2004], y opera mediante la aplicación de sustituciones de no terminales reiterativamente en las reglas de la gramática, a fin de eliminar dichos símbolos no terminales de dicha gramática y obtener una expresión regular formada exclusivamente por símbolos terminales de la misma. El algoritmo se describe en la Figura 4.4.1.

Entrada: Una gramática G
Salida: Una expresión regular equivalente a G, o bien un mensaje que informa de que G es no autoembebible
Método:

- Ordenar a los no terminales de G como A_1, …, A_n, siendo A_n el axioma de la gramática.
- Determinar expresiones regulares iniciales para cada A_i a partir de sus reglas $A_i ::= \alpha_{i,1}$, …, $A_i ::= \alpha_{i,k}$ de la forma: $\alpha_i = \alpha_{i,1} + \ldots + \alpha_{i,k}$ (α_i es la expresión regular de A_i).
- Refinar iterativamente dichas expresiones, desde i=1 hasta i=n:
  1. Se sustituye cada ocurrencia de cada no terminal A_j (j < i) en la expresión regular α_i para A_i por la correspondiente expresión regular de A_j.
  2. Se normaliza la expresión resultante de la forma: $A_i\alpha_R + \alpha_{NR} + \alpha_\lambda$, donde α_{NR} no tiene a A entre sus primeros, ni tampoco genera λ. Por su parte, α_λ es λ o bien ∅.
  3. Se normaliza la expresión resultante de la forma: $\alpha'_R A_i + \alpha'_{NR} + \alpha'_\lambda$, donde α'_{NR} no tiene a A entre sus últimos (es decir, no genera ninguna cadena de la forma αA), ni tampoco genera λ . Por su parte, α'_λ es λ o bien ∅.
  4. Si α_R , $\alpha'_R \neq \emptyset$ y α_R , $\alpha'_R \neq \lambda$ la gramática será no autoembebible (**terminar**).
  5. Si $\alpha_R \neq \emptyset$ y $\alpha_R \neq \lambda$, se sustituye α_i (la expresión para A_i) por $(\alpha_{NR} + \alpha_\lambda)\alpha_R{}^*$.
  6. Si no, si $\alpha'_R \neq \emptyset$ y $\alpha'_R \neq \lambda$, se substituye α_i (la expresión para A_i) por $\alpha'_R{}^*(\alpha'_{NR} + \alpha'_\lambda)$.
  7. Si no, si $\alpha_R = \lambda$ (también lo será α'_R), α_i (la expresión para A_i) se substituye por $(\alpha_{NR} + \alpha_\lambda)$.
  8. Una vez cerrada la expresión α_i, se substituye A_i por α_i en cada α_j de A_j (j < i).

- Si α_n contiene algún no terminal, la gramática será no autoembebible (**terminar**).
- La expresión regular buscada es α_n.

Figura 4.4.1. Método directo de traducción de una gramática no autoembebible a una expresión regular.

Normalización a izquierdas: $norm_L(\gamma, A)$
$norm_L(\emptyset, A)$ = $A\emptyset + \emptyset + \emptyset$
$norm_L(\lambda, A)$ = $A\emptyset + \emptyset + \lambda$
$norm_L(a, A)$ = $A\emptyset + a + \emptyset$ (a terminal)
$norm_L(A, A)$ = $A\lambda + \emptyset + \emptyset$
$norm_L(B, A)$ = $A\emptyset + B + \emptyset$ (B no terminal, B≠A)
$norm_L(\alpha^*, A)$ = $A\emptyset + \alpha^* + \emptyset$
$norm_L(\alpha+\beta, A)$ = $A(\alpha_R+\beta_R) + (\alpha_{NR}+\beta_{NR}) + (\alpha_\lambda+\beta_\lambda)$ (aplicando $norm_L(\alpha, A)=A\alpha_R+\alpha_{NR}+\alpha_\lambda$ y $norm_L(\beta, A)=A\beta_R+\beta_{NR}+\beta_\lambda$)
$norm_L(\alpha\beta, A)$ = $A\alpha_R\beta + \alpha_{NR}\beta + \emptyset$ (si α_λ es equivalente a ∅) | $A(\alpha_R\beta+\beta_R) + (\alpha_{NR}\beta+\beta_{NR}) + \beta_\lambda$ (si α_λ es equivalente a λ) (aplicando $norm_L(\alpha, A)=A\alpha_R+\alpha_{NR}+\alpha_\lambda$ y $norm_L(\beta, A)=A\beta_R+\beta_{NR}+\beta_\lambda$)

Normalización a derechas: $norm_R(\gamma, A)$
$norm_R(\emptyset, A)$ = $\emptyset A + \emptyset + \emptyset$
$norm_R(\lambda, A)$ = $\emptyset A + \emptyset + \lambda$
$norm_R(a, A)$ = $\emptyset A + a + \emptyset$ (a terminal)
$norm_R(A, A)$ = $\lambda A + \emptyset + \emptyset$
$norm_R(B, A)$ = $\emptyset A + B + \emptyset$ (B no terminal, B≠A)
$norm_R(\alpha^*, A)$ = $\emptyset A + \alpha^* + \emptyset$
$norm_R(\alpha+\beta, A)$ = $(\alpha_R+\beta_R)A + (\alpha_{NR}+\beta_{NR}) + (\alpha_\lambda+\beta_\lambda)$ (aplicando $norm_R(\alpha, A)=\alpha_R A+\alpha_{NR}+\alpha_\lambda$ y $norm_R(\beta, A)=\beta_R A+\beta_{NR}+\beta_\lambda$)
$norm_R(\alpha\beta, A)$ = $\alpha\beta_R A + \alpha\beta_{NR} + \emptyset$ (si β_λ es equivalente a ∅) | $(\alpha_R + \alpha\beta_R)A + (\alpha_{NR} + \alpha\beta_{NR}) + \alpha_\lambda$ (si β_λ es equivalente a λ) (aplicando $norm_R(\alpha, A)=\alpha_R A+\alpha_{NR}+\alpha_\lambda$ y $norm_R(\beta, A)=\beta_R A+\beta_{NR}+\beta_\lambda$)

Figura 4.4.2. Reglas de normalización de una expresión regular.

Respecto a las reglas de normalización utilizadas por el método, estas se detallan en la Figura 4.4.2, empleando $norm_L(\alpha,A)$ para normalizar α en la forma $A\alpha_R + \alpha_{NR} + \alpha_\lambda$ y, $norm_R(\alpha,A)$ para normalizarla en la forma $\alpha_R A + \alpha_{NR} + \alpha_\lambda$. En la implementación de las mismas se asume, así mismo, que las expresiones equivalentes a ∅ se reescriben a ∅, y las equivalentes a λ se reescriben a λ.

El correcto funcionamiento del algoritmo para gramáticas no autoembebibles se fundamenta en las siguientes observaciones:

- Las transformaciones que se aplican son correctas. Esto es evidente en el caso de la sustitución y de las reglas de normalización. En los pasos número 5 y 6 del algoritmo, la corrección se sustenta en los resultados de la teoría de las ecuaciones entre lenguajes [Leiss 1997]. El paso número 7 elimina una derivación cíclica de la forma $A \Rightarrow A$, lo que preserva la equivalencia.

- Partiendo, entonces, del supuesto de que la gramática de entrada es no autoembebible, se preserva el invariante de que, tras cada iteración i, α_i no contiene ningún A_j ($j \leq i$). Efectivamente, si se supone que la propiedad se cumple para todo $k < i$, tras realizar la primera sustitución, α_i no contendrá ningún A_j ($j < i$). Tras realizar la normalización, los términos resultantes deben estar libres de A_i, ya que, si no, la gramática resultará ser autoembebible. Por tanto, el único A_i que sobrevive se elimina en alguno de los pasos 5 a 7, por lo que el α_i resultante estará libre de A_i. El paso final de sustitución hacia atrás prueba el invariante. Como consecuencia, cuando el algoritmo termina, si la gramática es no autoembebible, en α_n habrá una expresión regular cerrada (sin símbolos no terminales), equivalente al lenguaje generado por la gramática.

Método directo

	Gramática BNF (axioma E) y expresiones iniciales	i = 1	i = 2	i = 3	i = 4	i = 5
1	**RE ::= a T RE | λ** α_1 = a T RE + λ	1. α_1 = a T RE + λ 2. RE ∅ + a T RE + λ 3. a T RE + ∅ + λ 6. α_1 = **(a T)***	1. α_1 = **(a T)*** 8. α_1 = **(a F RT)***	8. α_1 = **(a F (m F)*)***	8. α_1 = **(a(id+e) (m(id+e))*)***	
2	**T ::= F RT** α_2 = F RT		1. α_2 = F RT 2. T ∅+ F RT+∅ 3. ∅ T+ F RT+∅	8. α_2 = F **(m F)***	8. α_2 = **(id+e) (m(id+e))***	
3	**RT ::= m T | λ** α_3 = m T + λ			1. α_3 = m **F RT** + λ 2. RT ∅ + ∅ + λ 3. m **F** RT + ∅ + λ 6. α_3 = **(m F)***	8. α_3 = **(m(id+e))***	
4	**F ::= id | e** α_4 = id + e				1. α_4 = id + e 2. F ∅ + (id+e) +∅ 3. ∅ F + (id+e) +∅	
5	**E ::= T RE** α_5 = T RE					1. α_5 = **(id+e)(m(id+e))* (a(id+e)(m(id+e))*)*** 2. E ∅ + α_5 + ∅ 3. ∅ E + α_5 + ∅

Figura 4.4.3. Iteraciones y pasos del método directo para la obtención de una expresión regular a partir de una gramática no autoembebible BNF.

La Figura 4.4.3 muestra las iteraciones y pasos que se realizarían para obtener la expresión regular equivalente de la gramática BNF presentada en la columna más a la izquierda, según la ordenación de los símbolos no terminales que se ha elegido arbitrariamente. La principal desventaja del método queda reflejada en el orden en que son tratados los no terminales, ya que esto puede dar lugar a muchas sustituciones redundantes, o incluso, a una expresión regular mucho más extensa. Por ejemplo, de haberse ordenado los no terminales como: 1=F, 2=T, 3= RT, 4= RE, y 5 =E, se habrían realizado menos sustituciones de estos símbolos (6 sustituciones frente a las 9 de la figura anterior).

4.4.2 Método optimizado

El método optimizado refina el método directo (sección 4.4.1), evitando realizar sustituciones redundantes. Opera con una gramática no autoembebible G mediante tres fases consecutivas:

- Fase de normalización. Con cada símbolo no terminal A de G se asocian dos expresiones regulares equivalente al lenguaje generado: una de la forma $A\alpha R + \alpha NR + \alpha\lambda$, y otra de la forma $\alpha'RA + \alpha'NR + \alpha'\lambda$, que explicitan la recursión a izquierdas/a derechas del símbolo no terminal. Esta fase se detalla en la sección 4.4.2.1.

- Fase de cierre. Se asocia una única expresión αA a cada no terminal A, eliminando la parte recursiva a izquierdas/a derechas, si procede, mediante la aplicación del cierre de Kleene. Esta fase se detalla en la sección 4.4.2.2.

- Fase de expansión. Se sustituyen, bajo demanda, los no terminales en las expresiones por sus definiciones. Esta fase se detalla en la sección 4.4.2.3.

De esta forma, el método evita explícitamente las sustituciones redundantes mediante la realización de las sustituciones necesarias bajo demanda, una vez que se han eliminado las correspondientes recursiones *a izquierdas/a derechas*. Respecto al resto, las operaciones realizadas, son equivalentes a las del método directo para una ordenación adecuada de los no terminales (salvo que dicha ordenación no se fija a priori, sino que es descubierta, bajo demanda, en la fase de expansión).

4.4.2.1 Fase de normalización

La fase de normalización produce dos expresiones regulares de la forma $\alpha_{A\text{-}L} = A\alpha_R + \alpha_{NR} + \alpha_\lambda$ y $\alpha_{A\text{-}R} = \alpha'_R A + \alpha'_{NR} + \alpha'_\lambda$ para cada no terminal A de la gramática. La estrategia que se sigue consiste en realizar únicamente las expansiones de los símbolos no terminales más a la izquierda/más a la derecha, siguiendo para ello, las reglas de la gramática de partida. Utiliza un conjunto *visitados* para evitar ciclos, que, inicialmente, se fija a $\varnothing$. Dicha estrategia aplica los métodos de normalización de la Figura 4.4.4 que permiten obtener las expresiones

normalizadas para cada no terminal A como $\alpha_{A\text{-}L} = norm_L(\alpha_1 + \ldots + \alpha_k, A)$ y $\alpha'_{A\text{-}R} = norm_R(\alpha_1 + \ldots + \alpha_k, A)$ a partir de las reglas de A (A ::= α_1, …, A ::= α_k).

Normalización a izquierdas: $norm_L(\gamma, A)$	Normalización a derechas: $norm_R(\gamma, A)$
γ es: $\varnothing \rightarrow$ **devolver** $A\varnothing + \varnothing + \varnothing$. $\lambda \rightarrow$ **devolver** $A\varnothing + \varnothing + \lambda$. *a* (*a* terminal) $\rightarrow$ **devolver** $A\varnothing + a + \varnothing$. A $\rightarrow$ **devolver** $A\lambda + \varnothing + \varnothing$. B (B no terminal, B ≠ A) $\rightarrow$ si B contenido en visitados: - **devolver** $A\varnothing + B + \varnothing$. si no, si existe α_{LB} (la normalización para B está ya determinada): - **devolver** α_{LB}. si no: (1) - $norm_L(\beta_1 + \ldots + \beta_k, A) \rightarrow \alpha_{LB}$ (determinar la normalización para B respecto A, a partir de sus reglas B ::= β_1, …, B ::= β_k, y con visitados := visitados ∪ {B}). (2) - si α_{LB} es $A\varnothing + \alpha_{NR} + \alpha_\lambda$: $\alpha_{LB} = A\varnothing + B + \varnothing$. (re-determinar la normalización en términos de B) - devolver α_{LB}. $\alpha + \beta \rightarrow$ - $norm_L(\alpha, A) \rightarrow A\alpha_R + \alpha_{NR} + \alpha_\lambda$. - $norm_L(\beta, A) \rightarrow A\beta_R + \beta_{NR} + \beta_\lambda$. - **devolver** $A(\alpha_R+\beta_R) + (\alpha_{NR}+\beta_{NR}) + (\alpha_\lambda+\beta_\lambda)$. $\alpha\beta \rightarrow$ - $norm_L(\alpha, A) \rightarrow A\alpha_R + \alpha_{NR} + \alpha_\lambda$. - si α_λ es equivalente a $\varnothing$: **devolver** $A\alpha_R\beta + \alpha_{NR}\beta + \alpha_\lambda$. sino: - $norm_L(\beta, A) \rightarrow A\beta_R + \beta_{NR} + \beta_\lambda$. - **devolver** $A(\alpha_R\beta+\beta_R) + (\alpha_{NR}\beta+\beta_{NR}) + \beta_\lambda$.	γ es: $\varnothing \rightarrow$ **devolver** $\varnothing A + \varnothing + \varnothing$. $\lambda \rightarrow$ **devolver** $\varnothing A + \varnothing + \lambda$. *a* (*a* terminal) $\rightarrow$ **devolver** $\varnothing A + a + \varnothing$. A $\rightarrow$ **devolver** $\lambda A + \varnothing + \varnothing$. B (B no terminal, B ≠ A) $\rightarrow$ si B contenido en visitados: - **devolver** $\varnothing A + B + \varnothing$. si no, si existe α_{RB} (la normalización para B está ya determinada): - **devolver** α_{RB}. si no: (1) - $norm_R(\beta_1 + \ldots + \beta_k, A) \rightarrow \alpha_{RB}$ (determinar la normalización para B respecto A, a partir de sus reglas B ::= β_1, …, B ::= β_k, y con visitados := visitados ∪ {B}). (2) - si α_{RB} es $\varnothing A + \alpha_{NR} + \alpha_\lambda$: $\alpha_{RB} = \varnothing A + B + \varnothing$. (re-determinar la normalización en términos de B) - **devolver** α_{RB}. $\alpha + \beta \rightarrow$ - $norm_R(\alpha, A) \rightarrow \alpha_R A + \alpha_{NR} + \alpha_\lambda$. - $norm_R(\beta, A) \rightarrow \beta_R A + \beta_{NR} + \beta_\lambda$. - **devolver** $(\alpha_R+\beta_R)A + (\alpha_{NR}+\beta_{NR}) + (\alpha_\lambda+\beta_\lambda)$. $\alpha\beta \rightarrow$ - $norm_R(\beta, A) \rightarrow \beta_R A + \beta_{NR} + \beta_\lambda$. - si β_λ es equivalente a $\varnothing$: **devolver** $\alpha\beta_R A + \alpha\beta_{NR} + \beta_\lambda$. sino: - $norm_R(\alpha, A) \rightarrow \alpha_R A + \alpha_{NR} + \alpha_\lambda$. - **devolver** $(\alpha_R+\alpha\beta_R)A + (\alpha_{NR}+\alpha\beta_{NR}) + \alpha_\lambda$.

Figura 4.4.4. Métodos optimizados para la normalización de un no terminal A a izquierdas/a derechas.

Los métodos de normalización a izquierdas/a derechas están directamente relacionados con las reglas de normalización del método directo (Figura 4.4.2). Obsérvese que, en la normalización de la expresión para un no terminal A, no se expanden las definiciones de aquellos no terminales que no producen a A en posiciones más a la izquierda/más a la derecha (2). Como consecuencia, se acortan en tamaño considerablemente las expresiones regulares resultantes frente al método directo, pues dichos no terminales embebidos posteriormente se reemplazarán por sus definiciones resultantes de aplicar la fase de cierre (por lo tanto, más reducidas). Por otro lado, aunque el caso $\varnothing$ no es estrictamente necesario, se contempla para facilitar la generalización del método en la sección 4.6.1. El caso α^* no se contempla debido a que estos casos nunca aparecerán en la gramática original (al estar en formato BNF), ni se aplicarán, en este punto, reglas de transformación para la eliminación de la parte recursiva de alguna expresión (respecto al método directo, pasos número 5 y 6, sección 4.4.1). Por último, la función que desempeñan α_{LB} y α_{RB} en (1) es permitir reutilizar normalizaciones a fin de evitar realizar cómputos redundantes. Así mismo, dado que las expresiones regulares pueden representarse mediante árboles de sintaxis abstracta, esto permite también compartir

estructura y reducir la memoria ocupada por las expresiones (las estructuras resultantes serán grafos acíclicos en lugar de árboles, como muestra la Figura 4.4.5).

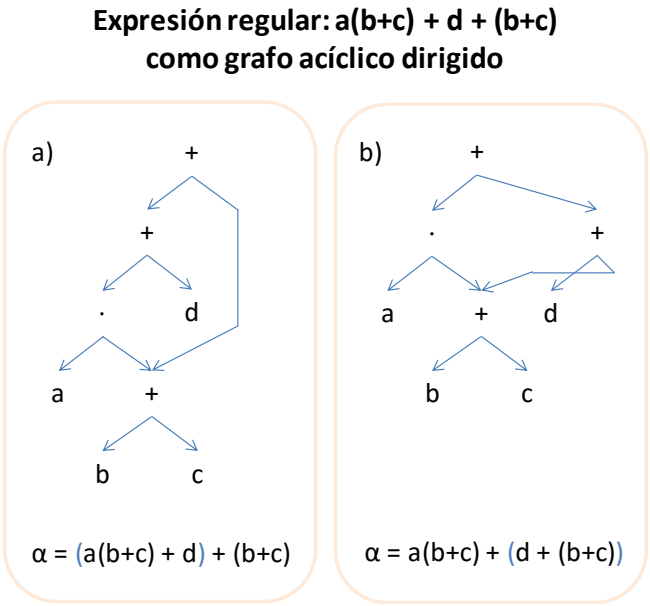


Figura 4.4.5. Expresiones regulares como grafos acíclicos dirigidos.

Por último, a medida que se obtienen las expresiones regulares normalizadas $\alpha_{A\text{-}L} = A\alpha_R + \alpha_{NR} + \alpha_\lambda$ y $\alpha_{A\text{-}R} = \alpha'_R A + \alpha'_{NR} + \alpha'_\lambda$ para un no terminal A de la gramática, el proceso puede terminar inmediatamente si no se cumple la condición de gramática no autoembebible, por darse el caso $\alpha_R, \alpha'_R \neq \varnothing$ y $\alpha_R, \alpha'_R \neq \lambda$. La Figura 4.4.6 muestra un ejemplo de la fase de normalización.

Fase de normalización y fase de cierre

Gramática BNF y expresiones iniciales	$norm_L(\alpha_A, A)$	$norm_R(\alpha_A, A)$	Cierre
E ::= T RE α_E = T RE	$\alpha_{E\text{-}L}$ = E∅ + T RE + ∅	$\alpha_{E\text{-}R}$ = ∅E + T RE + ∅	3. α_E = T RE
RE ::= a T RE | λ α_{RE} = a T RE + λ	$\alpha_{RE\text{-}L}$ = RE∅ + a T RE + λ	$\alpha_{RE\text{-}R}$ = a T RE + ∅ + λ	2. α_{RE} = (a T)*
T ::= F RT α_T = F RT	$\alpha_{T\text{-}L}$ = T∅ + F RT + ∅	$\alpha_{T\text{-}R}$ = F m T + F + ∅	2. α_T = (F m)* F
RT ::= m T | λ α_{RT} = m T + λ	$\alpha_{RT\text{-}L}$ = RT∅ + m T + λ	$\alpha_{RT\text{-}R}$ = m F RT + ∅ + λ	2. α_{RT} = (m F)*
F ::= id | e α_F = id + e	$\alpha_{F\text{-}L}$ = F∅ + (id+e) + ∅	$\alpha_{F\text{-}R}$ = ∅F + (id+e) + ∅	3. α_F = id+e

Figura 4.4.6. Normalizaciones obtenidas por la fase de normalización para la gramática no autoembebible que figura en la columna más a la izquierda, y cierre aplicado a las normalizaciones según sección 4.4.2.2.

4.4.2.2 Fase de cierre

Esta fase determina una sola expresión regular α_A a partir de las dos expresiones regulares normalizadas $\alpha_{A\text{-}L} = A\alpha_R + \alpha_{NR} + \alpha_\lambda$ y $\alpha_{A\text{-}R} = \alpha'_R A + \alpha'_{NR} + \alpha'_\lambda$ de cada no terminal A de la gramática. Para ello, la estrategia que sigue es similar a la del método directo, aplicando un análisis de casos para eliminar la recursión explícita a izquierdas/a derechas. Los casos son los siguientes:

1. Si $\alpha_R \neq \varnothing$ y $\alpha_R \neq \lambda$ entonces $\alpha_A = (\alpha_{NR} + \alpha_\lambda)\alpha_R^*$.
2. Si no, si $\alpha'_R \neq \varnothing$ y $\alpha'_R \neq \lambda$ entonces $\alpha_A = \alpha'_R{}^*(\alpha'_{NR} + \alpha'_\lambda)$.
3. Si no, $\alpha_A = \alpha_{NR} + \alpha_\lambda$.

La Figura 4.4.6 muestra un ejemplo de la fase de cierre, en la columna más a la derecha.

4.4.2.3 Fase de expansión

Expansión: expand(γ)

γ es:
- λ → **devolver** λ.
- *a* (*a* terminal) → **devolver** *a*.
- A (A no terminal) →
  - si A no contenido en visitados:
    - si α'_A (la expresión expandida para A está ya determinada):
      - **devolver** α'_A.
    - si no:
    (1) - expand(α_A) → α'_A (determinar la expresión expandida para A, a partir de su expresión sin expandir α_A, con visitados := visitados ∪ {A}).
      - **devolver** α'_A.
  - si no: **terminar y decidir:** *la gramática es autoembebible* (ciclo detectado).
- $\alpha + \beta$ →
  - expand(α) → α'.
  - expand(β) → β'.
  - **devolver** $\alpha' + \beta'$.
- $\alpha\beta$ →
  - expand(α) → α'.
  - expand(β) → β'.
  - **devolver** $\alpha'\beta'$.
- α^* →
  - expand(α) → α'.
  - **devolver** α'^*.

Figura 4.4.7. Método de expansión para una expresión regular con posibles símbolos no terminales embebidos.

En esta fase se realiza una expansión bajo demanda de las expresiones regulares obtenidas en la fase de cierre, partiendo de la expresión α_S correspondiente al no terminal S axioma de la gramática. Como resultado, o bien se obtiene una la expresión equivalente a la gramática α'_S, o bien se decide que la gramática es no autoembebible (en caso de que en el proceso de expansión se descubra un ciclo). La estrategia que se sigue se detalla en la Figura 4.4.7, donde se utiliza un conjunto de no terminales *visitados* (inicialmente con {S}) para detectar ciclos.

Fase de expansión

Gramática BNF y expresiones tras el cierre	Expansión
E ::= T RE α_E = T RE	α_E = ((id+e) m)* (id+e) (a ((id+e) m)* (id+e))*
RE ::= a T RE | λ α_{RE} = **(a T)***	α_{RE} = (a ((id+e) m)* (id+e))*
T ::= F RT α_T = **(F m)* F**	α_T = ((id+e) m)* (id+e)
RT ::= m T | λ α_{RT} = **(m F)***	(sin expandir) α_{RT} = (m F)*
F ::= id | e α_F = id+e	α_F = id+e

Figura 4.4.8. Expresiones regulares finales obtenidas de realizar la fase de expansión a partir de la expresión para el axioma, tras el cierre, de la gramática no autoembebible.

Por último, las definiciones expandidas asociadas a los símbolos no terminales (1) pueden reutilizarse, y así evitar recomputaciones innecesarias, permitiendo, de nuevo compartir estructura y reducir la memoria ocupada por las expresiones. La Figura 4.4.8 muestra un ejemplo de la fase de expansión, donde la expresión regular asociada al axioma de la gramática es el resultado final (celdas en rojo).

4.5 Configuración del método de comprobación de la conformidad

Las secciones anteriores ponen en evidencia que existen múltiples alternativas para configurar el método de comprobación de la conformidad. Efectivamente:

- La transformación de las expresiones regulares en autómatas finitos deterministas puede llevarse a cabo, bien aplicando el método directo de las derivadas, que genera autómatas deterministas, bien aplicando alguno de los otros métodos analizados (método de Thompson, método de Berry-Sethi, método de las derivadas parciales), y, a continuación, la construcción por subconjuntos.

- Por su parte, la transformación de gramáticas no autoembebibles en autómatas finitos deterministas puede llevarse a cabo, bien aplicando el algoritmo de Nederhof, seguido de la construcción por subconjuntos, bien aplicando el método de conversión a expresiones regulares que hemos desarrollado en esta tesis (sección 4.4), seguido de alguno de los métodos de conversión a autómatas finitos deterministas esbozados anteriormente.

El *diagrama de características* [Czarnecki & Eisenecker 2000] de la Figura 4.5.1 resume, por tanto, posibles formas de configurar el método de comprobación de la conformidad (en rojo se muestra una posible configuración, consistente en aplicar el método de Thompson seguido de la construcción por subconjuntos para transformar expresiones regulares en autómatas deterministas, y el algoritmo de Nederhof junto con la construcción por subconjuntos para transformar gramáticas no autoembebibles en autómatas). Por tanto, restringiéndonos únicamente a los métodos más representativos, revisados en esta tesis, junto al método de conversión de gramáticas no autoembebibles a expresiones regulares que hemos desarrollado, tenemos 20 posibles alternativas para configurar el método de comprobación de la conformidad.

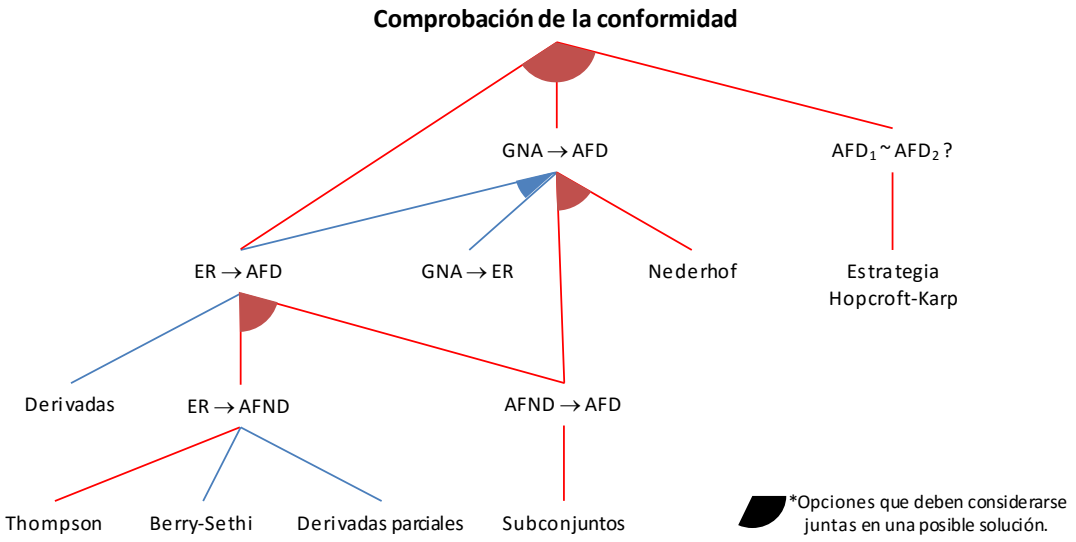


Figura 4.5.1. Diagrama de características que representa posibles formas de configurar el método de comprobación de la conformidad. En rojo, una posible configuración.

Para elegir, de entre todas las alternativas posibles, la más adecuada, hemos estudiado los siguientes ejes de variabilidad:

- Complejidad en tiempo y en memoria de las soluciones. La solución más adecuada debería presentar buen comportamiento en cuanto a tiempo de ejecución y memoria requerida.

- Pereza de las construcciones. Dado que el método depende, en última instancia, de la comprobación de la equivalencia entre autómatas finitos deterministas, en caso de descubrir que dos autómatas no son equivalentes, no será necesario explorar todos sus estados. Por tanto, es deseable que dichos estados se expandan de manera perezosa, conforme se requieran en el procedimiento de refutación.

- Facilidades para el diagnóstico de la no equivalencia. Los usuarios finales del método son desarrolladores humanos que modelizan estructuras documentales mediante gramáticas específicas para los procesamientos. Por tanto, en caso de que estas gramáticas no sean conformes con las gramáticas documentales de partida, será conveniente ofrecer una información útil para diagnosticar la causa de la no conformidad.

A continuación, se analizan cada uno de estos ejes de variabilidad, lo que nos permitirá, por último, elegir un método de comprobación de la conformidad concreto.

4.5.1 Complejidad

El factor principal de la complejidad del método de comprobación de la conformidad está en la obtención de los autómatas finitos deterministas asociados a las expresiones regulares y a las gramáticas no autoembebibles. De esta forma, todos los métodos disponibles exhiben complejidades asintóticas exponenciales en el peor de los casos [Leiss 1981]. En concreto:

- En el caso del método de las derivadas, comprobamos que, en la práctica dicha complejidad teórica no depende de casos *extraños* y *artificiales*, sino que puede surgir con cierta frecuencia. Para ello llevamos a cabo la realización de distintas simulaciones con expresiones regulares generadas aleatoriamente y con un algoritmo equipado de una estrategia de normalización de expresiones estándar, en base a las reglas ACI y a otras reglas usuales (sección 4.3.1). En base a dichas simulaciones pudimos comprobar que, aproximadamente en 1 de entre 10 casos, se generaba una expresión para la que el algoritmo, simplemente, corría fuera de memoria. Esto es debido, sobre todo, a que, para ciertas expresiones (especialmente las que contienen muchas iteraciones anidadas), el algoritmo puede generar una gran cantidad de derivadas, de tamaños muy grandes.

- Para el resto de los métodos, la complejidad reside en la construcción por subconjuntos.

De esta forma, es pertinente tener en cuenta el número de estados de los AFNDs generados por los distintos métodos de conversión de expresiones regulares a AFNDs, ya que dicho número incidirá, posteriormente, en el número de estados del AFD, y, por tanto, en el comportamiento práctico del método de determinización, y, por ende, en el de comprobación de la equivalencia. A este respecto, y en lo que se refiere a los algoritmos de conversión de expresiones regulares en AFNDs:

- Tal y como se demuestra en [Champarnaud & Ziadi 2002], el AFND construido mediante el método de las derivadas parciales es, en realidad, una compactación del autómata de las posiciones obtenido mediante el método de Berry-Sethi, en el sentido de que cada derivada parcial puede representar un subconjunto de posiciones equivalentes entre sí.

- Por su parte, tal y como se demuestra en [Ilie & Yu 2003], el resultado de eliminar las λ-transiciones del AFND de Thompson (resultado que se denomina *autómata de seguidores*), es, así mismo, una compactación del autómata de las posiciones.

Por tanto, tanto el autómata de las derivadas parciales como el autómata de los seguidores tendrán siempre, a lo sumo, tantos estados como el autómata de las posiciones (aunque también pueden tener menos, hecho que ocurre, por ejemplo, cuando se genera la misma derivada parcial por distintos caminos, lo que a su vez se encuentra estrechamente vinculado con la ocurrencia de una misma subexpresión varias veces en una expresión, lo que puede ocurrir habitualmente en la práctica). Dichos autómatas no son, en sí, comparables respecto al número de estados, aunque es posible aplicar propuestas que, como las descritas en [García et al. 2011], fusionan ambos métodos para generar autómatas con, a lo sumo, tantos estados como el de las derivadas parciales y el de seguidores. Por otra parte, como se evidencia en [García et al. 2011, Ilie & Yu 2003, Brüggemann-Klein 1993], todos estos autómatas (posiciones, derivadas parciales, seguidores, etc.) pueden construirse mediante algoritmos con complejidad cuadrática con respecto al tamaño de la expresión regular, por lo que el coste de la construcción del AFND en sí no supone un factor determinante. De hecho, y desde un punto de vista práctico, en la realización de las simulaciones aludidas anteriormente, las distintas combinaciones de métodos se comportaron de manera apropiada para todos los casos, al contrario que el de las derivadas. Así mismo, en dicha simulación, y para cada algoritmo, siempre encontramos casos en los que dicho algoritmo se comportó mejor que el resto.

En relación con la transformación de gramáticas no autoembebibles en AFDs, en el caso más habitual, en el que exista únicamente un no terminal en cada nivel de recursión, hemos comprobado que el algoritmo de Nederhof construye un AFND similar al construido por el método de Thompson a partir de la expresión regular generada por nuestro método de traducción a expresiones regulares. De hecho, desde un punto de vista práctico, comprobamos mediante simulación que, tampoco en este caso los métodos son comparables de cara al coste final del proceso de determinización (siempre encontramos casos en los que uno se comportó mejor que otro, independientemente del método empleado para la traducción de expresiones regulares a autómatas en combinación con nuestro método de traducción de gramáticas).

4.5.2 Pereza

En relación con los distintos métodos de transformación de expresiones regulares a AFDs:

- Es evidente que el algoritmo de las derivadas posibilita una implementación perezosa directa. Efectivamente, cada estado del autómata tendrá asociada una derivada, que puede derivarse en el momento en el que se soliciten transiciones aún no computadas para dicho estado.

- Por su parte, también es evidente que la construcción por subconjuntos puede implementarse perezosamente, de forma que los estados se construyan conforme se requieran, la primera vez que se consulte el destino de alguna de las transiciones que llegan a los mismos.

Por otra parte, y en relación con los distintos métodos de transformación de expresiones regulares a AFNDs:

- Aunque el método de Thompson, en primera instancia, no parece susceptible de admitir una implementación perezosa, el proceso de eliminación de las λ-transiciones (que conduce, como ya se ha indicado anteriormente, al autómata de seguidores, que, a su vez, es una compactación del autómata de las posiciones) *sí* puede implementarse perezosamente. En este proceso, es posible representar explícitamente las transiciones que van expandiéndose, a fin de evitar aplicaciones redundantes de la operación de λ-cierre, lo que implica, a su vez, la construcción perezosa del autómata de seguidores.

- El método de Berry-Sethi no es, sin embargo, susceptible de ser implementado perezosamente.

- Por último, el método basado en derivadas parciales soporta claramente una implementación perezosa, ya que cada estado contiene la información necesaria para producir los estados a los que puede transitar: la expresión regular que representa la derivada asociada a dicho estado, y que, a su vez, puede volver a derivarse para obtener los estados siguientes. Así mismo, como en el caso de la construcción perezosa del autómata de seguidores, es posible representar explícitamente las transiciones que van expandiéndose, a fin de evitar derivaciones redundantes. Esta incrementabilidad no es fácilmente extrapolable, sin embargo, a la optimización descrita en [Champarnaud & Ziadi 2002].

De esta forma, tanto el método de Thompson como el método de las derivadas parciales, combinados con una implementación perezosa de la construcción por subconjuntos, son susceptibles de soportar una traducción perezosa de expresiones regulares en autómatas deterministas, que soporte, además, la construcción perezosa de los AFNDs subyacentes. Dicha traducción consistirá, en todos los casos, de tres fases bien diferenciadas:

- Traducción de la expresión regular a una representación abstracta (un árbol de sintaxis abstracta en el caso del método de las derivadas parciales, un AFND con λ-transiciones en el caso del método de Thompson). Esta fase es, en todos los casos, impaciente (es decir, se realizará completamente, independientemente de donde, finalmente, el proceso de comprobación de la equivalencia tenga éxito, o fracase).

- Traducción perezosa de la representación abstracta en un AFND sin λ-transiciones. En el caso del método de las derivadas parciales, dicha traducción se llevará a cabo, perezosamente, por derivación. En el caso del método de Thompson, dicha etapa se llevará a cabo mediante una aplicación bajo demanda de la operación de λ-cierre. En todos los casos, se construirá bajo demanda el AFND resultante, a fin de evitar derivaciones o aplicaciones del λ-cierre redundantes.

- Traducción perezosa del AFND resultante de la fase anterior a un AFD. Para ello se aplicará una implementación perezosa simplificada de la construcción por subconjuntos, en la que puede obviarse el λ-cierre, al trabajar, siempre, sobre AFNDs sin λ-transiciones.

En relación con la traducción de gramáticas no autoembebibles a AFNDs, en ambos casos es posible formular satisfactoriamente estrategias perezosas. Para ello, se sustituirá la primera fase en el procedimiento anterior (traducción a una estructura abstracta) por la aplicación sobre la gramática del algoritmo de Nederhof en un caso, y de nuestro algoritmo de traducción a expresiones regulares en el otro. Dichos algoritmos se aplicarán de manera impaciente (es decir, independientemente del resultado de comprobación de la equivalencia, se construirá completamente el AFND con λ-transiciones en un caso, y la expresión regular equivalente en el otro).

4.5.3 Facilidad para el diagnóstico

Tal y como se ha comentado en la sección 4.3.2, el algoritmo para la comprobación de la equivalencia de dos AFDs permite obtener fácilmente, en caso de no equivalencia de los autómatas, una cadena testigo que demuestra dicha no equivalencia. No obstante, sería necesario, además, contar con información adicional que pueda ayudar al desarrollador.

Desde este punto de vista, en lo que respecta al autómata generado a partir de la expresión regular en la gramática EBNF, relativa al modelo de contenidos, la aplicación de los métodos de las derivadas y de las derivadas parciales son, claramente, ventajosos sobre el de Thompson o el de Berry-Sethi, ya que permiten informar de manera directa *del lenguaje que resta por reconocer para cada estado*. Efectivamente, dado un estado *Q* del AFD:

- En el caso de haber aplicado el algoritmo de las derivadas, dicho lenguaje vendrá descrito por la derivada asociada a *Q*.

- En el caso de haber aplicado el algoritmo de las derivadas parciales, el lenguaje vendrá descrito por la suma de las derivadas parciales en dicho estado.

En el caso de los métodos de Thompson o de Berry-Sethi, la obtención directa de dicha información no es, sin embargo, factible, ya que los estados no deterministas carecen de dicha información.

En el caso de la traducción de gramáticas no autoembebibles, nuestro método de traducción a expresiones regulares ofrece claras ventajas frente al de Nederhof en lo que se refiere a la facilidad de diagnóstico. Efectivamente, mientras que la combinación de nuestro algoritmo con el método de las derivadas o el de las derivadas parciales nos permite ofrecer información sobre el conjunto de cadenas que falta por reconocer para cada estado (en forma de una expresión regular), esto no es cierto para el autómata de Nederhof (la información no está presente en los estados no deterministas).

4.5.4 Método resultante

El análisis realizado en los puntos anteriores nos permite configurar racionalmente una solución para la comprobación de la conformidad (Figura 4.5.2). Dicha solución emplea nuestro método de traducción de gramáticas no autoembebibles a expresiones regulares, y el método de las derivadas parciales para transformar expresiones regulares en autómatas. Efectivamente:

- El método de las derivadas parciales es adecuado desde el punto de vista de la facilidad de diagnóstico y de la construcción perezosa de los autómatas implicados. Desde un punto de vista de eficiencia, es claramente superior al método de las derivadas. Así mismo, genera AFNDs que nunca exceden en tamaño a los generados por el método de Berry-Sethi, y no necesariamente peores en tamaño que los autómatas de seguidores (resultantes de eliminar las λ-transiciones de los generados por el método de Thompson). Así mismo, las complejidades asintóticas de construcción de los AFNDs son similares a las de los otros métodos. En nuestras simulaciones hemos comprobado, así mismo, que dicho resultado sigue siendo válido en la práctica, aún en el caso perezoso.

- Nuestro método de traducción de gramáticas no autoembebibles a expresiones regulares es más adecuado, desde el punto de vista de facilidad de diagnóstico, que el método de Nederhof. Por otra parte, en los casos más habituales, exhibe una eficiencia similar al de Nederhof, y, así mismo, posibilita una estrategia de construcción perezosa análoga al mismo.

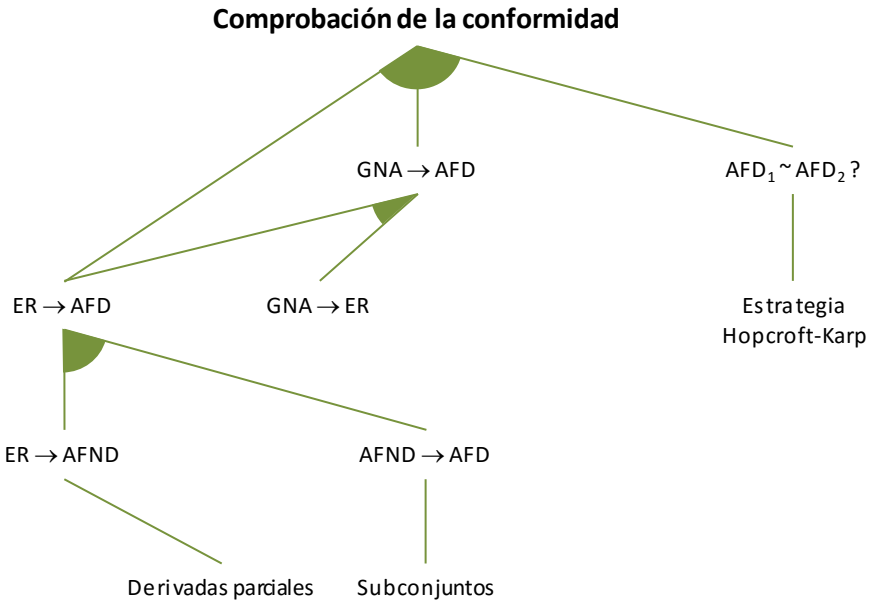


Figura 4.5.2. Solución final para la comprobación de la conformidad.

De esta forma, el método resultante configurado en la Figura 4.5.2:

- Comprobará, sucesivamente, la equivalencia de cada subgramática no autoembebible asociada con cada no terminal de núcleo con la correspondiente expresión regular en la gramática EBNF que abstrae a la gramática documental.

- Dicha comprobación de equivalencia entre gramáticas no autoembebibles y expresiones regulares implementará, de manera perezosa la construcción de todos los autómatas involucrados (tanto los AFNDs como los AFDs), de manera que dicha construcción opere bajo demanda, conforme se requiera desde la estrategia por refutación para la prueba de la equivalencia basada en el método de Hopcroft-Karp (este hecho, el de la construcción perezosa del AFND, supone una ventaja frente a propuestas similares, para la comprobación de la equivalencia de expresiones regulares, como la de [Almeida et al. 2010]).

- La estrategia basada en el método de Hopcroft-Karp llevará cuenta, así mismo, de las distintas cadenas que conducen a plantear cada objetivo Q~Q'.

- En caso de detectar la falsedad de un objetivo Q~Q' en la prueba de la equivalencia relativa a un no terminal de núcleo *A*, el método informará de la no conformidad debida al no terminal *A*, justificada por la correspondiente cadena que conduce a dicho objetivo, y por las expresiones regulares asociadas con Q y con Q' (que representan, en ambos casos, los lenguajes que quedan por reconocer).

Entrada: ER y ER', cuya equivalencia o no equivalencia quiere determinarse.
Salida: Las posibles salidas son:

- **equivalentes**
- (α,*D*,*D*'), donde *D* y *D*' son informaciones de diagnóstico de los siguientes tipos: (i) **no final** δE, con δE una derivada, (ii) **final** con δE una derivada, (iii) **error**

Método:
- Obtener el AFD equivalente a ER y el AFD' equivalente a ER'
- Para cada estado *Q* de AFD y de AFD', determinar la clase de equivalencia de *Q* como {*Q*}.
- Sean Q_0 y Q'_0 los estados iniciales de AFD y AFD' respectivamente:
  - Si Q_0 es final y Q'_0 no lo es
    - **terminar con** (λ,**final** ER, **no final** ER')
  - Si Q'$_0$ es final y Q_0 no lo es
    - **terminar con** (λ,**no final** ER, **final** ER')
  - Marcar el objetivo $\lambda|\text{-}Q_0 \sim Q'_0$ como a explorar y fijar la clase de equivalencia de Q_0 y de Q'$_0$ a {Q_0, Q'_0}.
- Mientras haya objetivos a explorar**:**
  - Elegir uno de estos objetivos $\alpha|\text{-}Q \sim Q'$. Desmarcarlo como objetivo a explorar
  - Para cada símbolo *X*:
    - Considerar las transiciones $Q \text{ –}X\text{-> } P$ y $Q' \text{ –}X\text{-> } P'$.
      - Si $\delta P \neq \varnothing$, $\delta P' = \varnothing$
        - Si *P* es final
          - **terminar con** (αX, **final** δP, **error**)
        - Si *P* es no final
          - **terminar con** (αX, **no final** δP, **error**)
      - Si $\delta P = \varnothing$, $\delta P' \neq \varnothing$
        - Si *P'* es final
          - **terminar con** (αX, **error**, **final** $\delta P'$)
        - Si *P'* es no final
          - **terminar con** (αX, **error, no final** $\delta P'$)
      - Si P es final, P' es no final
        - **terminar con** (αX, **final** δP, **no final** $\delta P'$)
      - Si *P'* es final, P es no final
        - **terminar con** (αX, **no final** δP, **final** $\delta P'$)
    - Si la clase de equivalencia de *P* es distinta de la clase de equivalencia de *P'*
      - Marcar el *objetivo* $\alpha X \text{ |- } P \sim P'$ como objetivo a explorar.
      - Sea Γ la clase de equivalencia de *P* y Γ' la clase de equivalencia de *P'*. Para cada estado *R* en $\Gamma \cup \Gamma'$ fijar la clase de equivalencia de *R* a $\Gamma \cup \Gamma'$.
- **Decidir:** las expresiones ***son equivalentes***.

Figura 4.5.3. Estrategia de comprobación de la equivalencia basada en el método de Hopcroft-Karp.

La estrategia de comprobación de la equivalencia basada en el método de Hopcroft-Karp, por su parte, refina dicho método para:

- Por una parte, detectar condiciones de no equivalencia en caso de que por uno de los autómatas se alcance un estado de error (representado por una expresión regular asociada $\varnothing$).

- Por otra parte, en caso de detectar la no equivalencia, devolver información precisa para el diagnóstico. Dicha información es del tipo (α,*D*,*D*'), donde: (i) α es, como ya se ha indicado anteriormente, la cadena testigo que ha conducido a plantear el objetivo que ha violado las condiciones de equivalencia, (ii) *D* y *D*' son información de diagnóstico relativas a los estados de dicho objetivo (más concretamente, **final** *E*, en caso de que el correspondiente estado sea final, **error**, en caso de que el correspondiente estado sea un estado de error, o **no final** *E*, en caso de que el correspondiente estado sea no final; tanto en el caso de estado final como de estado no final, se informa, además, de la correspondiente expresión regular asociada al estado). Esta información permite, por tanto, ofrecer mensajes de diagnóstico significativos.

La Figura 4.5.3 muestra el pseudocódigo de la estrategia de comprobación de la equivalencia, basada en el método de Hopcroft-Karp, resultante. En dicho pseudocódigo, mediante δQ se denota la disyunción de las derivadas parciales asociadas al estado *Q*. Este hecho evidencia la manera de explotar dicha información para posibilitar una mejor información de diagnóstico. Así mismo, los objetivos mantenidos por esta estrategia son, ahora, de la forma α|-*Q*~*Q*', donde α representa la cadena que conduce a plantear la equivalencia *Q*~*Q*'.

4.6 Estudio de la utilidad del método de comprobación de la conformidad

El objetivo de esta sección es proporcionar una evidencia empírica sobre la utilidad práctica del método para facilitar la formulación de gramáticas específicas para el procesamiento. Para ello, fue necesario adaptarse a las características específicas de la comunidad con la que podíamos realizar las pruebas: alumnos de segundo ciclo de Ingeniería Informática, de la asignatura de Procesadores de Lenguaje en la Facultad de Informática de la Universidad Complutense de Madrid. Dado que en dicha asignatura no se trabaja con gramáticas EBNF, sino con gramáticas BNF, las cuales los alumnos tienen que transformar para cumplir ciertos requisitos, fue necesario generalizar el método de transformación de gramáticas no autoembebibles en expresiones regulares para que pudiese soportar gramáticas arbitrarias, tal y como se detalla en la sección 4.6.1. Como resultado es posible obtener una gramática EBNF equivalente a la gramática BNF de entrada. Esto permite comprobar la conformidad de una gramática BNF respecto a otra de partida:

- Transformando ambas en gramáticas EBNF.
- Comprobando que comparten los mismos símbolos no terminales (símbolos *de núcleo*).
- Comprobando la equivalencia de las expresiones regulares que definen cada símbolo *de núcleo*.

El resultado de esta generalización es una herramienta bautizada como *Grammar Equivalence Checker*, y que aproxima de manera heurística la comprobación de la equivalencia de dos gramáticas incontextuales (su respuesta es afirmativa si la comprobación de la conformidad esbozada tiene éxito, ya que dicha comprobación es una condición más fuerte que la equivalencia, o inconcluyente en otro caso). En la sección 4.6.2 se muestra en detalle el funcionamiento y las características de dicha herramienta.

Mientras que la herramienta no realiza exactamente el método de comprobación de la conformidad, al generalizar a éste, sí permite extrapolar los resultados empíricos de evaluación de la utilidad al primero (método que, al ser completo, ofrece menos dificultades de uso que el generalizado). Gracias a ello, se consiguió llevar a cabo la mencionada prueba empírica sobre un escenario real, que se analiza en la sección 4.6.3, obteniendo unos resultados muy positivos sobre la utilidad práctica del método de comprobación de la conformidad.

4.6.1 Generalización del método optimizado de transformación de gramáticas no autoembebibles a otros tipos de gramáticas

Durante el desarrollo del método optimizado de transformación de gramáticas no autoembebibles a expresiones regulares (sección 4.4.2), surgió la posibilidad de su modificación para dar soporte a gramáticas arbitrarias. Si bien el método optimizado permite obtener una expresión regular correspondiente al lenguaje derivado por un símbolo no terminal de la gramática de entrada, es posible obtener en el caso de que éste lenguaje no sea regular, una expresión EBNF en función de otros símbolos de la gramática. Como resultado, se obtiene una gramática EBNF equivalente a la original, en lugar de una expresión regular asociada con el axioma de la gramática. Si la gramática es no autoembebible, se obtiene una gramática EBNF, con una única regla (tras una limpieza) para el axioma sin símbolos no terminales en su cuerpo y equivalente a la expresión regular obtenible por el método optimizado. Para conseguir este resultado, el método optimizado se modifica de la siguiente manera:

- En la fase de normalización, se asocia con cada no terminal A, en vez de dos expresiones de la forma $A\alpha_R + \alpha_{NR} + \alpha_\lambda$ y $\alpha'_R A + \alpha'_{NR} + \alpha'_\lambda$, una única expresión α_A de la forma $\alpha_A = A\gamma A + A\alpha + \beta A + \delta$. Para su obtención, se realizan consecutivamente los siguientes pasos:

    1. Se normaliza por la izquierda la expresión para A: $norm_L(\alpha_1 + \ldots + \alpha_k, A) \rightarrow A\boldsymbol{\alpha_R} + \boldsymbol{\alpha_{NR}} + \boldsymbol{\alpha_\lambda}$, a partir de las reglas de A ($A ::= \alpha_1, \ldots, A ::= \alpha_k$).

    2. Se normalizan las expresiones anteriores por la derecha (excepto α_λ, pues no puede contener a A):

- $norm_R(\alpha_R, A) \rightarrow \beta_R A + \beta_{NR} + \beta_\lambda$.
- $norm_R(\alpha_{NR}, A) \rightarrow \gamma_R A + \gamma_{NR} + \gamma_\lambda$.

3. Se normaliza finalmente la expresión α_A para A de la siguiente forma:

   - $A(\beta_R A+\beta_{NR}+\beta_\lambda) + \gamma_R A+\gamma_{NR}+\gamma_\lambda + \alpha_\lambda = A\beta_R A + A\beta_{NR} + A\beta_\lambda + \gamma_R A + \gamma_{NR} + \gamma_\lambda + \alpha_\lambda = A\beta_R A + A(\beta_{NR}+\beta_\lambda) + \gamma_R A + (\gamma_{NR}+\gamma_\lambda+\alpha_\lambda)$.

Fase de normalización

Gramática BNF y expresiones iniciales	$norm_L(\alpha_A,A)$	$norm_R(\alpha_{A\text{-}L.R},A)$	$norm_R(\alpha_{A\text{-}L.NR},A)$	Normalización final
E ::= T RE α_E = T RE	$\alpha_{E\text{-}L}$ = E∅ + T RE + ∅	$\beta_{E\text{-}R}$ = ∅E + ∅ + ∅	$\gamma_{E\text{-}R}$ = ∅E + T RE + ∅	α_E = E∅E + E∅ + ∅E + T RE
RE ::= a T RE | λ α_{RE} = a T RE + λ	$\alpha_{RE\text{-}L}$ = RE∅ + a T RE + λ	$\beta_{RE\text{-}R}$ = ∅RE + ∅ + ∅	$\gamma_{RE\text{-}R}$ = a T RE + ∅ + ∅	α_{RE} = RE∅RE + RE∅ + a T RE + λ
T ::= F RT α_T = F RT	$\alpha_{T\text{-}L}$ = T∅ + F RT + ∅	$\beta_{T\text{-}R}$ = ∅T + ∅ + ∅	$\gamma_{T\text{-}R}$ = F m T + F + ∅	α_T = T∅T + T∅ + F m T + F
RT ::= m T | λ α_{RT} = m T + λ	$\alpha_{RT\text{-}L}$ = RT∅ + m T + λ	$\beta_{RT\text{-}R}$ = ∅RT + ∅ + ∅	$\gamma_{RT\text{-}R}$ = m F RT + ∅ + ∅	α_{RT} = RT∅RT + RT∅ + m F RT + λ
F ::= id | o E c α_F = id + e	$\alpha_{F\text{-}L}$ = F∅ + (id+o E c) + ∅	$\beta_{F\text{-}R}$ = ∅F + ∅ + ∅	$\gamma_{F\text{-}R}$ = ∅F + (id+o E c) + ∅	α_F = F∅F + F∅ + ∅F +(id+o E c)

Figura 4.6.1. Normalizaciones intermedias y finales, obtenidas por la fase de normalización para la gramática BNF que figura en la columna más a la izquierda.

- En la fase de cierre, se aplica la expresión siguiente para transformar α_A: AγA + Aα + βA + δ $\Rightarrow$ (β*δα*γ)* β*δα*. Esta es una expresión bien conocida de la teoría de las ecuaciones entre lenguajes [Leiss 1999], que preserva la equivalencia y que permite eliminar las ocurrencias de A en los extremos de la expresión normalizada.

Fase de cierre y fase de expansión

Gramática BNF y expresiones normalizadas	Cierre	Expansión
E ::= T RE α_E = E∅E + E∅ + ∅E + T RE	α_E = T RE	α_E = ((id+o E c) m)* (id+o E c) (a ((id+o E c) m)* (id+o E c))* **E → ((id|o E c) m)* (id|o E c) (a ((id|o E c) m)* (id|o E c))***
RE ::= a T RE | λ α_{RE} = RE∅RE + RE∅ + a T RE + λ	α_{RE} = (a T)*	α_{RE} = (a ((id+o E c) m)* (id+o E c))*
T ::= F RT α_T = T∅T + T∅ + F m T + F	α_T = (F m)* F	α_T = ((id+o E c) m)* (id+o E c)
RT ::= m T | λ α_{RT} = RT∅RT + RT∅ + m F RT + λ	α_{RT} = (m F)*	(sin expandir) α_{RT} = (m F)*
F ::= id | o E c α_F = F∅F + F∅ + ∅F +(id+o E c)	α_F = id+o E c	α_F = id+o E c

Figura 4.6.2. Expresiones regulares finales y gramática EBNF obtenida de realizar la fase de cierre y, posteriormente, de expansión, a partir de las expresiones normalizadas de la gramática BNF.

- En la fase de expansión, los ciclos detectan los no terminales de la gramática EBNF resultante (por lo tanto, la expresión devuelta es, simplemente, el no terminal que produce el ciclo). Estos no terminales no se expanden por sus definiciones (las definiciones en sí pueden almacenarse y consultarse directamente para evitar cómputos redundantes).

En la Figura 4.6.1 y Figura 4.6.2 se muestra el proceso de transformación de una gramática BNF a una gramática EBNF (celdas verdes, tras una limpieza) mediante las tres fases del método generalizado.

4.6.2 La herramienta Grammar Equivalence Checker

Grammar Equivalence Checker es una herramienta que fue desarrollada en este trabajo de tesis para facilitar la elaboración y refinamiento de gramáticas incontextuales y automatizar el proceso de comprobación de conformidad entre ellas, extendidas al caso general: gramáticas BNF arbitrarias. En su interior, se han implementado todos los métodos expuestos a lo largo de este capítulo, y se rige por los mismos criterios conceptuales y el mismo enfoque de desarrollo. En la sección 4.6.2.1 se describe su lenguaje de especificación, y en la sección 4.6.2.2 el uso y funcionamiento de la herramienta.

4.6.2.1 Lenguaje de especificación

Las especificaciones gramaticales en esta herramienta siguen el formalismo básico descriptivo BNF. De esta forma:

- Las producciones se escriben utilizando la notación habitual (A ::= α), siendo posible agrupar varias producciones para un mismo no terminal, separando los cuerpos por | (A ::= α_0 |…|α_n). Las producciones se terminan en ";" (A ::= α_0 |…|α_n;). Así mismo, la cadena vacía λ no tiene representación simbólica, sino que se identifica con la ausencia de cuerpo de una producción (p.e., A ::= ; representa la regla A ::= λ).
- Los no terminales son *palabras* (cadenas de caracteres que empieza por una letra o guion bajo seguida de una secuencia de letras, guiones bajos o números) que figuran como cabeza de una producción.
- Por su parte, los terminales son palabras que no aparecen como cabeza de producción. También se admite como no terminal cualquier cadena de caracteres o símbolos especiales escritos entre comillas simples, como, por ejemplo: *'*&%'* o *'terminal'*.
- Por último, el axioma será el primer no terminal que figure como cabeza de la primera producción de la gramática.

4.6.2.2 Uso y funcionamiento

A la hora de realizar la comprobación de la conformidad de dos gramáticas BNF en la herramienta, éstas se transforman, utilizando el método generalizado (sección 4.6.1), en gramáticas EBNF. Seguidamente se comprueba que ambas contienen los mismos no terminales. Por último, para cada no terminal, se comprueba la equivalencia de las correspondientes expresiones regulares, en ambas gramáticas, utilizando el método de las derivadas parciales para transformarlas perezosamente a AFNDs, la construcción por subconjuntos para generar perezosamente los AFDs correspondientes, y la estrategia basada en el método por refutación de Hopcroft-Karp para probar la equivalencia en sí.

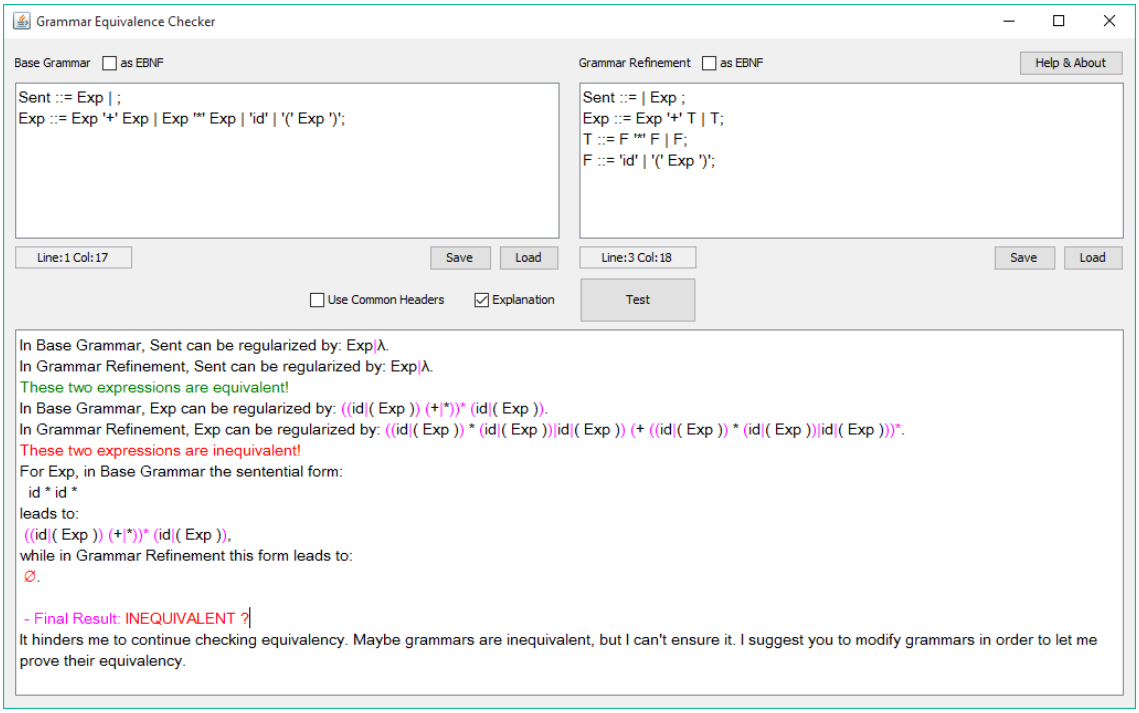


Figura 4.6.3. Interfaz de la herramienta Grammar Equivalence Checker.

La Figura 4.6.3 muestra la interfaz de *Grammar Equivalence Checker* junto a un ejemplo de funcionamiento. En esta interfaz:

- *Base Grammar* hace referencia a la gramática de partida (el análogo a la gramática EBNF que abstrae la gramática documental en el enfoque descrito en la sección 4.2).
- *Grammar Refinement* hace referencia al refinamiento de la gramática de partida (el análogo a la gramática específica para el procesamiento).

La herramienta, entonces, comprueba la conformidad de ambas especificaciones. mostrando, si se solicita, una explicación del resultado. En caso de que la regularización de las gramáticas (es decir, el resultado de aplicar el algoritmo de transformación a EBNF) produzca símbolos de núcleo distintos, la herramienta informa de dicho hecho, así como de su imposibilidad para decidir la equivalencia. En otro caso, procede con la comprobación de la conformidad:

- Mostrando, para cada símbolo de núcleo, las expresiones regulares que lo definen en cada gramática.
- En caso de detectarse una no equivalencia, mostrando: (i) la cadena testigo que conduce a los estados probados no equivalentes, y (ii) las correspondientes expresiones regulares que definen los lenguajes aceptados a partir de dichos estados (sumas de las correspondientes derivadas parciales).

Una vez finalizado dicho proceso, en caso de que todas las expresiones hayan resultado ser equivalentes, la herramienta anunciará la equivalencia de las especificaciones de partida. En otro caso, informará que no es posible decidir sobre dicha equivalencia. De esta forma, la herramienta proporciona una solución incompleta al problema de comprobación de la equivalencia, basado en la conformidad de gramáticas desarrollada en esta tesis.

4.6.3 Experiencia de evaluación

A fin de evaluar la utilidad y eficacia, en la práctica, del proceso de comprobación de conformidad entre gramáticas, se utilizó *Grammar Equivalence Checker* en una experiencia en un escenario real. Esta experiencia permitió a un grupo de alumnos profundizar en la materia de Procesadores de Lenguaje, y su vez, nos brindó la posibilidad de realizar un análisis crítico del proceso de refinamiento y comprobación de la conformidad entre gramáticas desde el punto de vista externo.

Para la realización de la experiencia, se seleccionó un grupo de veinte alumnos de la asignatura de Procesadores de Lenguaje de los estudios de Ingeniería en Informática de la Facultad de Informática de la Universidad Complutense de Madrid, durante el curso académico 2013-2014. Se les expuso previamente en qué consistía el carácter indecidible de equivalencia entre gramáticas incontextuales [Bar-Hillel et al. 1961], y cómo la herramienta *Grammar Equivalence Checker* podría serles útil en el desarrollo de ejercicios típicos de la asignatura, consistentes en refinar gramáticas incontextuales de base en otras ajustadas a la implementación de los analizadores. Establecida la motivación, a los alumnos se les permitió, como ayuda adicional, el uso de apuntes u otros documentos para la resolución de los ejercicios expuestos en la Figura 4.6.4.

A fin de obtener conclusiones más objetivas, se propuso la creación de dos grupos: diez alumnos que realizarían el ejercicio con la ayuda de la herramienta (grupo A, gramáticas 1 y 2) y diez alumnos que realizarían el ejercicio de manera tradicional, con lápiz y papel (grupo B, gramáticas 1 y 3). Se partió de la hipótesis de que el ejercicio planteado, mostrado en la Figura 4.6.4, era adecuado al nivel de conocimiento que poseían los alumnos. Los ejercicios se resuelven aplicando transformaciones conocidas que fueron impartidas en clases teóricas y prácticas previamente. De esta manera, la finalidad del ejercicio es obtener una gramática equivalente a la original, que presente las características necesarias para ser soportada por un analizador de tipo descendente predictivo con un símbolo de preanálisis, es decir, una gramática LL(1).

	Acondiciona las siguientes gramáticas para permitir un análisis descendente predictivo recursivo	**Solución**
Gramática 1	E ::= T * E | T / E | T T ::= T + F | T – F | F F ::= n | (E)	E ::= T RE RE ::= + T RE | - T RE | λ T ::= F RT RT ::= * T | / F | λ F ::= n | (E)
Gramática 2	E ::= T * E | T / T | T T ::= T + F | F – F | F F ::= id | id (Ps) | id ( ) | (E) Ps ::= Ps , E Ps ::= E	E ::= T RE RE ::= * E | / T | λ T ::= F RT0 RT1 RT1 ::= + F RT1 |λ RT0 ::= - F | λ F ::= id RF | (E) RF ::= (RF2) | λ RF2 ::= Ps | Ps ::= E RPs RPs ::= , E RPs RPs ::= λ
Gramática 3	S ::= id | id : { Es } | id : { } Es ::= Es , E Es ::= E E ::= E + T | T - T | T T ::= F * T | F / F | F F ::= id | n | (E)	S ::= id RS RS ::= : { RS2 | λ RS2 ::= Es } | } Es ::= E REs REs ::= , E REs REs ::= λ E ::= T RE0 RE1 RE0 ::= - T | λ RE1 ::= + T RE1 | λ T ::= F RT RT ::= * T | / F | λ F ::= id | n | (E)

Figura 4.6.4. Ejercicios propuestos para la experiencia educativa.

Para la realización de los ejercicios, a los grupos A y B se les otorgó el tiempo de una hora. La solución a los ejercicios debía quedar escrita y entregada en papel para su corrección y evaluación posterior. Una vez terminado el ejercicio, al grupo B (sin herramienta) se le propuso realizar el mismo ejercicio propuesto al grupo A en sus hogares, pero con la herramienta, a fin de obtener una crítica más diversa. La Figura 4.6.5 muestra la puntuación obtenida (eje vertical, escala 0 a 10) en los ejercicios realizados por los alumnos (eje horizontal, nº de alumnos) pertenecientes al grupo A (rojo) y B (azul):

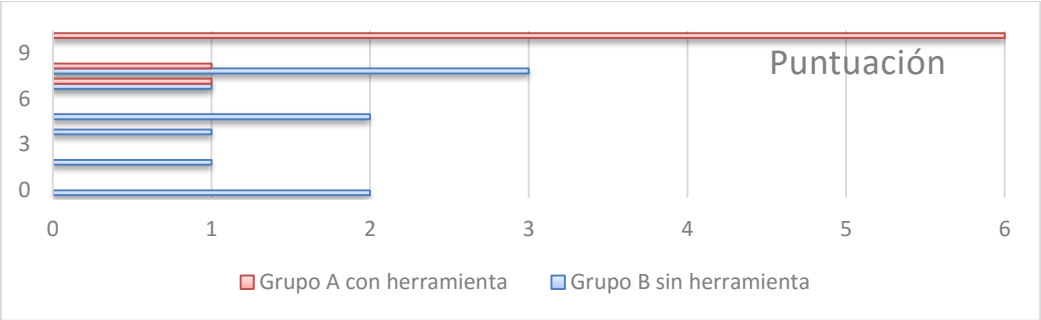


Figura 4.6.5. Puntuación obtenida por los alumnos.

Analizando los resultados obtenidos de la Figura 4.6.5, el grupo A (con herramienta) obtuvo puntuaciones muy superiores al grupo B (sin herramienta). Aunque el número de asistentes del grupo A se redujo a ocho personas, en conjunto obtuvieron una puntuación media de 9.375 puntos frente a los 4.7 puntos obtenidos por los diez asistentes del grupo B. La mayoría de los alumnos del grupo A (6 de 8 personas) obtuvieron la puntuación máxima de 10, mientras que la máxima nota obtenida por los alumnos del grupo B fue de 8 puntos (3/10 personas). Estos resultados proporcionan evidencia positiva de que el proceso de refinamiento y comprobación de la conformidad entre gramáticas incontextuales utilizando la herramienta *Grammar Equivalence Checker* tiene una funcionalidad real muy satisfactoria.

De forma inmediatamente posterior a la realización del ejercicio, se planteó a cada alumno rellenar un cuestionario de satisfacción. A los alumnos del grupo B (sin herramienta), como ya se expuso anteriormente, se les planteó realizar los ejercicios que fueron propuestos al grupo A con el uso de la herramienta para que los resolvieran en sus hogares, y rellenasen el cuestionario de satisfacción. El cuestionario de satisfacción se compone de dos partes: un cuestionario sobre el ejercicio en sí (Figura 4.6.6), y un cuestionario sobre la utilidad de realizar el proceso de refinamiento y comprobación de la conformidad entre gramáticas con la herramienta *Grammar Equivalence Checker* (Figura 4.6.7).

La Figura 4.6.6 muestra la valoración promedio, en una escala de 0 a 5 en el eje horizontal, que los 18 alumnos asistentes (8 del grupo A y 10 del grupo B) han respondido a cada una de las cuestiones planteadas en el eje vertical. Analizando los resultados obtenidos, el nivel del ejercicio planteado realmente era adecuado, como se estableció en la hipótesis de partida, al nivel de conocimientos de los alumnos para la resolución de los ejercicios propuestos. Cabe destacar que la mayoría de los alumnos mostraron un elevado interés por disponer de una herramienta que les permitiese comprobar la corrección del ejercicio realizado, en otras palabras, una herramienta que les permita comprobar si la gramática que han desarrollado y refinado es equivalente a la original.

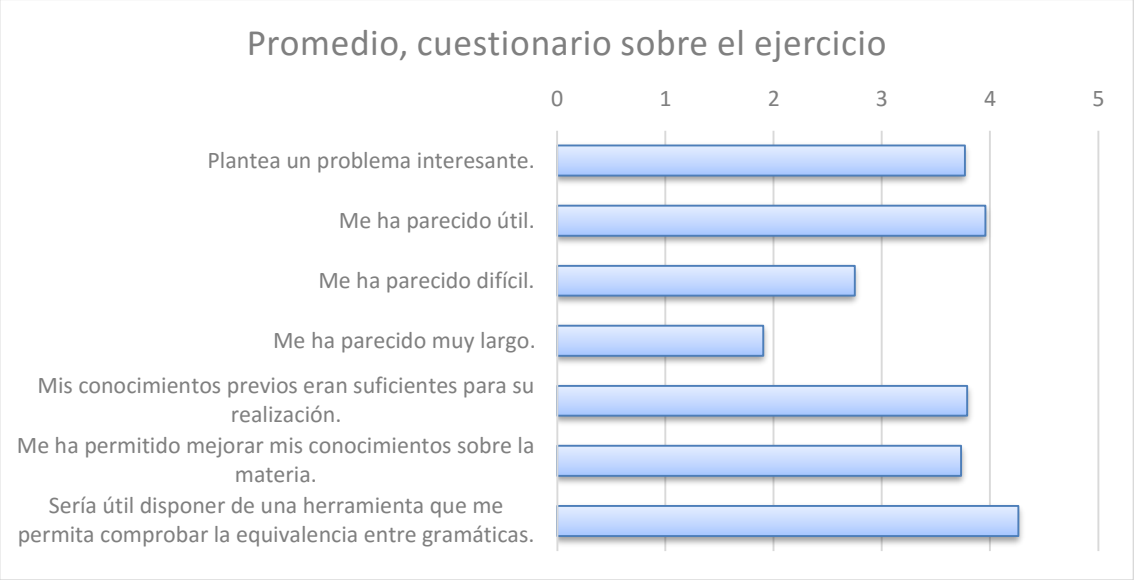


Figura 4.6.6. Cuestionario sobre el ejercicio.

La Figura 4.6.7 muestra la valoración promedio, en una escala de 0 a 5 en el eje horizontal, que 14 alumnos (8 del grupo A y 6 del grupo B desde sus hogares) han dado a cada una de las cuestiones planteadas en el eje vertical. Analizando los resultados obtenidos, junto con los comentarios aportados por los alumnos (cuestión opcional que no se incluye en la figura anterior), se concluye que la herramienta en la práctica es realmente útil, fácil e intuitiva de usar, y como herramienta cumple satisfactoriamente como asistente para realizar el refinamiento de gramáticas incontextuales y la comprobación de la conformidad entre dichas gramáticas. También se concluye, que la calidad de respuesta e información otorgada por la herramienta (y, por lo tanto, por el método planteado) es lo suficientemente útil y rica en información como para que los alumnos prefieran disponer de dicha herramienta durante sus estudios en la materia de la asignatura de Procesadores de Lenguaje.

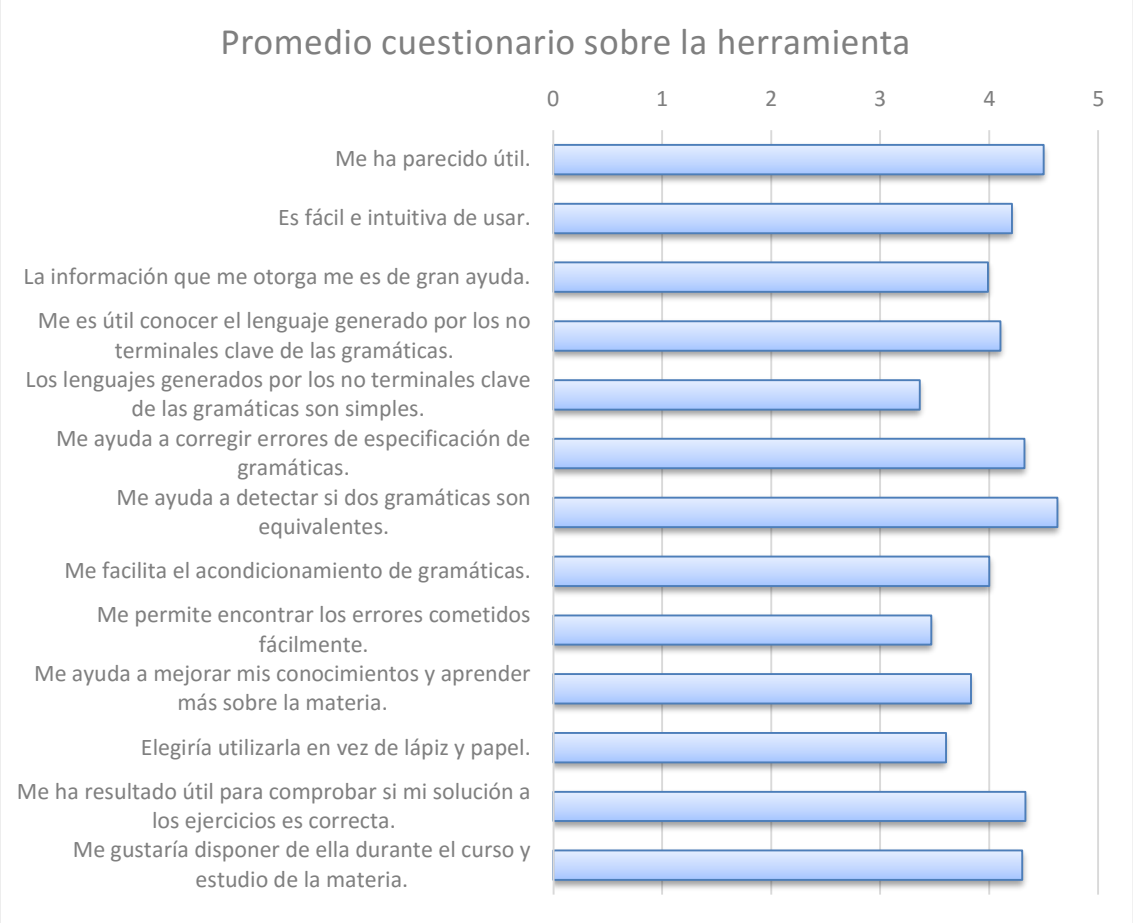


Figura 4.6.7. Cuestionario sobre el proceso de formulación de gramáticas y comprobación de la conformidad con la herramienta Grammar Equivalence Checker.

4.7 A modo de conclusión

En este capítulo se ha abordado la primera de las limitaciones de los enfoques dirigidos por lenguajes desarrollados en el grupo ILSA: la carencia de un método sistemático para formular gramáticas específicas para el procesamiento, así como de un método efectivo para la comprobación de la conformidad de dichas gramáticas con respecto a las gramáticas documentales. Para ello se ha comenzado desarrollando un método de formulación de gramáticas específicas para el procesamiento que:

- Comienza fomentando la formulación de una gramática EBNF de base que abstraiga las características estructurales básicas de la gramática documental.
- Promueve la obtención de la gramática específica para el procesamiento mediante la realización gramatical de las expresiones regulares que definen los distintos no terminales de núcleo de la gramática EBNF.

- Promueve, así mismo, el refinamiento iterativo, tanto de la gramática específica para el procesamiento como de la gramática EBNF de base.

La limitación de la clase de gramáticas que realizan expresiones regulares EBNF a la de las gramáticas no autoembebibles es, en cierta forma, lógica, ya que, por una parte, este tipo de gramáticas generan lenguajes regulares. Por otra parte, estas gramáticas no imponen restricciones específicas sobre la forma de las producciones, siempre y cuando se preserve la condición de no autoembebilidad. De esta forma, constituyen la manera más natural de modelar mediante reglas BNF un lenguaje regular.

Así mismo, el método de formulación de gramáticas específicas para el procesamiento descrito permite obtener un método efectivo de comprobación de la conformidad, que se reduce a comprobar la equivalencia de cada expresión regular definitoria de cada no terminal de núcleo en la gramática EBNF con respecto a la realización de la misma mediante una gramática no autoembebible. El método en sí es efectivo, ya que dicha comprobación se reduce a comprobar la equivalencia de los autómatas deterministas asociados con cada tipo de descriptor.

En concreto, en este capítulo se ha propuesto un algoritmo de comprobación de la conformidad que utiliza un método para comprobar la equivalencia entre una expresión regular y una gramática no autoembebible orientado a ofrecer buena información de diagnóstico en caso de que la comprobación sea negativa. Para ello, se generan autómatas deterministas cuyos estados están explícitamente etiquetados con descriptores de los lenguajes que quedan por reconocer una vez que estos son alcanzados. Dichos descriptores son conjuntos de expresiones regulares (el lenguaje en sí será la disyunción de dichas expresiones). Para ello: (i) se adopta el método de las derivadas parciales para traducir expresiones regulares en AFNDs equivalentes, (ii) se ha desarrollado un método novedoso para traducir gramáticas no autoembebibles en expresiones regulares equivalentes, que se puede conectar con el citado método de las derivadas parciales. Como consecuencia, cada estado de los AFNDs resultantes en ambos casos estarán etiquetados por expresiones regulares (derivadas parciales de la expresión original) que describirán el lenguaje que resta por reconocer en dicho estado. Por tanto, la determinización de los mismos por el método de los subconjuntos ofrece el resultado deseado.

Así mismo, la estrategia propuesta tiene la virtud de soportar una implementación con un alto grado de pereza, lo que, a su vez, es una característica deseable de cara a la comprobación de la equivalencia (ya que, en caso de encontrarse que las entradas no son equivalentes, usualmente no va a ser necesario expandir los autómatas en su totalidad). Efectivamente, no únicamente la construcción por subconjuntos puede realizarse de manera perezosa, sino también los propios AFNDs.

Por último, en este capítulo se ha puesto de manifiesto también cómo es posible generalizar el método de conversión de gramáticas no autoembebibles a gramáticas incontextuales arbitrarias. En este caso, el resultado no será ya una expresión regular, sino una gramática EBNF equivalente. Esta generalización nos ha permitido abordar la comprobación de la conformidad, no ya de gramática BNF con respecto a gramática EBNF de base, sino también entre dos gramáticas BNF arbitrarias. Como resultado hemos construido una herramienta denominada *Grammar Equivalence Checker*, y hemos evaluado la misma en el contexto de un curso de Procesadores de Lenguaje de último año de Ingeniería Informática, con el fin de tener evidencia respecto a la eficacia y usabilidad del método de comprobación de la conformidad. Los resultados obtenidos al respecto han sido, en ambos casos, positivos.

Capítulo 5
Gramáticas de Atributos Multivista para el Procesamiento Dirigido por Lenguajes de Documentos XML

5.1 Introducción

Las gramáticas de atributos multivista son gramáticas de atributos especiales que mejoran las capacidades de modularización del formalismo básico para el desarrollo dirigido por lenguajes de aplicaciones de procesamiento de documentos XML (Capítulo 3). La idea que subyace bajo el concepto multivista consiste en permitir descomponer una tarea de procesamiento compleja en tareas más simples, y especificar cada una de estas tareas utilizando fragmentos de gramáticas de atributos formulados sobre las estructuras sintácticas más convenientes. De esta forma, en las gramáticas de atributos multivista cada fragmento puede formularse sobre una gramática incontextual distinta. La propuesta resuelve, por tanto, la principal limitación del enfoque de descripción modular de tareas de procesamiento XML, mediante gramáticas de atributos, propuesto anteriormente en el grupo de investigación ILSA [Sarasa & Sierra 2013-b], en el que todos los fragmentos se formulaban sobre la misma gramática incontextual.

De esta forma, una primera aproximación al concepto multivista, puede entenderse como el hecho de unificar varias gramáticas de atributos diferentes, que describen distintos aspectos de una tarea de procesamiento más compleja. Como resultado de dicha unificación, se obtendrá una gramática de atributos global que permita realizar el procesamiento eficiente de la entrada mediante un único análisis y una única evaluación semántica. Los problemas que este concepto introducen no son triviales, sin embargo, ya que cada gramática de atributos que interviene en la composición puede estar formulada sobre una gramática incontextual diferente. De esta forma:

- Por una parte, cada una de las gramáticas de atributos que intervienen en la composición debe formularse sobre gramáticas incontextuales equivalentes entre sí, en el sentido de generar toda ellas el mismo lenguaje. Así mismo, dichas gramáticas no podrán elegirse de manera arbitraria, sino que deberán preservar algún tipo de característica estructural común que permita su interrelación en el resultado final de la composición. Estos dos aspectos pueden abordarse de manera unificada mediante el desarrollo realizado en el Capítulo 4.

- Por otra parte, la gramática incontextual subyacente a la unificación será necesariamente ambigua, ya que cada fragmento utilizado se formulará sobre una gramática incontextual potencialmente diferente, aunque equivalente al resto. Por tanto, será necesario utilizar algoritmos de análisis sintácticos más generales a los convencionales (basados en gramáticas LL(k), LR(k), LALR(k), etc.) que sean capaces de tratar adecuadamente dicha ambigüedad, y que, al mismo tiempo, sean capaces de producir estructuras sintácticas en las que se explicite la relación existente entre los diferentes fragmentos compuestos.

- Por último, será necesario disponer de un mecanismo descriptivo que permita especificar adecuadamente el nuevo tipo de gramáticas introducido.

En este capítulo se abordan todos estos aspectos. Se describe también XLOP3, una evolución del entorno XLOP para el desarrollo y mantenimiento de aplicaciones de procesamiento de documentos XML mediante gramáticas de atributos multivista. De esta forma, en la sección 5.2 se describen cómo son y cómo se construyen las especificaciones de gramáticas de atributos multivista. Seguidamente, en la sección 5.3 se detalla cómo realizar la fase de análisis de documentos XML respecto a estas gramáticas multivista. Posteriormente, en la sección 5.4 se detalla la fase de evaluación semántica inducida por estas gramáticas. La sección 5.5 presenta el entorno XLOP3, como meta-herramienta para el desarrollo dirigido por lenguajes de aplicaciones de procesamiento XML utilizando gramáticas de atributos multivista. Finalmente, la sección 5.6 presenta un caso de estudio completo consistente en el desarrollo con XLOP3 de una aplicación de procesamiento de documentos XML de complejidad no trivial. Como cierre de este capítulo, en la sección 5.7 se recogen las conclusiones alcanzadas.

En este capítulo se refina, desarrolla y detalla la propuesta inicial de gramáticas de atributos multivista realizada en [Temprado et al. 2010-a] como un enfoque a la modularización de especificaciones de tareas de procesamiento de documentos XML mediante gramáticas de atributos.

5.2 Gramáticas de atributos multivista

Las *gramáticas de atributos multivista* consisten en una colección de fragmentos de gramáticas de atributos definidos sobre un conjunto de gramáticas incontextuales, que pueden diferir entre sí, y conformes con una misma gramática EBNF (en el sentido descrito en el Capítulo 4, y, por tanto, equivalentes entre sí). Cada uno de estos fragmentos se denomina *vista gramatical*. Como resultado, la propuesta mejora el carácter modular de las realizadas en [Sarasa & Sierra 2013-b], ya que, por una parte, permite describir la tarea de procesamiento en base a aspectos más sencillos con fragmentos de gramáticas, y, por otra, no obliga a que dichos fragmentos se formulen sobre la misma gramática incontextual, sino que permite elegir una gramática específica para cada procesamiento particular abordado por cada vista gramatical.

Cuando las vistas gramaticales se unen para formar la gramática de atributos multivista, constituyen la especificación de múltiples tareas de procesamiento independientes entre sí, pero con la capacidad de poder comunicarse entre sí y aprovechar procesamiento en común. Esto es posible gracias a que cada tarea, especificada por cada vista gramatical, puede comunicarse con las demás a través de los atributos de los símbolos gramaticales compartidos entre las vistas. Dichos símbolos son los símbolos de la gramática EBNF de base, los denominados *símbolos de núcleo* en el Capítulo 4 (sección 4.2). Para ello, la unión de las vistas implica que, durante el procesamiento de cada entrada, los árboles de análisis sintácticos subyacentes a cada una de las vistas se fusionen en una única estructura: un *bosque de análisis sintáctico*. La fusión se realiza "pegando" los distintos árboles a través de los correspondientes símbolos de núcleo. Dicho bosque puede, entonces, decorarse con atributos y operaciones de cómputo que permiten realizar cada tarea de procesamiento y que establecen un flujo de datos que permite la comunicación entre las distintas tareas.

De esta forma, el desarrollo de una aplicación de procesamiento XML basado en gramáticas de atributos multivista implica la ejecución de los siguientes cuatro pasos:

- Planteamiento del problema. Esta actividad consiste en la descripción del problema, estableciendo cuales son las tareas de procesamiento que se deben realizar.

- Establecimiento de la gramática EBNF de base. Esta actividad se centra en el establecimiento de la gramática EBNF que sirva de base para la formulación de las distintas vistas gramaticales. Dicha gramática abstraerá la gramática documental del lenguaje de marcado en el sentido descrito en el Capítulo 4.

- Caracterización de la sintaxis de las vistas. Esta actividad consiste en la elaboración de las gramáticas incontextuales sobre las que se formularán cada una de las vistas gramaticales. Por tanto, cada una de estas gramáticas estarán específicamente orientadas a cada tarea de procesamiento.

- Caracterización de la semántica de las vistas. Esta actividad consiste en la caracterización de las tareas de procesamiento mediante el añadido de semántica a las estructuras gramaticales resultantes de la actividad anterior.

- Ejecución de la especificación. En esta actividad la especificación se transforma en un procesador de lenguaje operativo, que actúa sobre las entradas para llevar a cabo el procesamiento de las mismas.

La Figura 5.2.1 muestra el flujo de trabajo de esta metodología de desarrollo basada en gramáticas de atributos multivista. Obsérvese que posee un carácter iterativo e incremental, en el sentido de que siempre es posible volver a las actividades previas para introducir nuevos refinamientos.

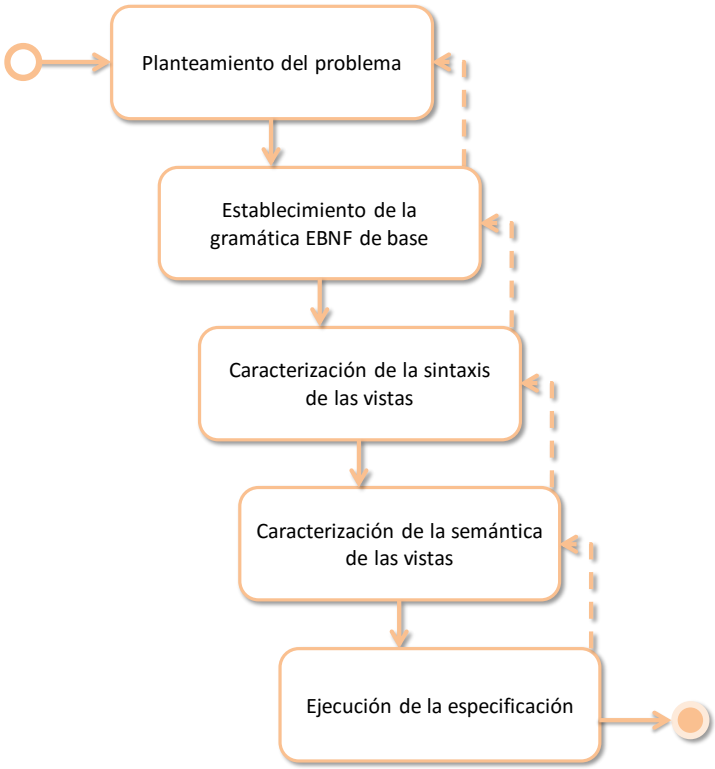


Figura 5.2.1. Secuencia de actividades propuesta como metodología de desarrollo de aplicaciones de procesamiento de documentos XML con gramáticas de atributos multivista.

5.2.1 Planteamiento del problema

La primera actividad en la formulación de una gramática multivista consiste en el planteamiento del problema de procesamiento a abordar. Dicho planteamiento implica decidir las tareas de procesamiento a realizar.

DTD

```
<!ENTITY % opnd '(Num|Var|p)'>
<!ENTITY % exp
'(%opnd;,((Add|Mul),%opnd;)*)'>

<!ELEMENT f (%exp;)>
<!ELEMENT p (%exp;)>
<!ELEMENT Add EMPTY>
<!ELEMENT Mul EMPTY>
<!ELEMENT Num (#PCDATA)>
<!ELEMENT Var (#PCDATA)>
```

Instancia XML

```
<f>
   <Var>x</Var>
   <Mul/>
   <p>
      <Num>2</Num>
      <Add/>
      <Num>3</Num>
   </p>
</f>
```

Figura 5.2.2. DTD e instancia XML para las fórmulas aritméticas.

A modo de ejemplo, consideraremos un procesamiento consistente en evaluar las fórmulas aritméticas representadas por el tipo de documentos de entrada que se muestra en la Figura 5.2.2. Como la DTD de la Figura 5.2.2 hace patente, dichas fórmulas son una versión muy simplificada de las fórmulas aritméticas de la matemática tradicional (en concreto, la instancia XML de la Figura 5.2.2 codifica la fórmula aritmética siguiente: "x*(2+3)"). Tales fórmulas consisten en operaciones cuyos operandos son números (<Num>) y variables (<Var>), utilizando únicamente los operadores de suma (<Add>) y multiplicación (<Mul>). Adicionalmente se pueden agrupar explícitamente operaciones, describiendo subfórmulas entre paréntesis (<p>). Las variables, a diferencia de los números, que son valores obtenidos directamente del documento procesado, son nombres que refieren a un valor, y que deben ser determinados por el usuario en tiempo de ejecución. Quedando determinado el valor de las variables por el usuario, las formulas se podrán calcular automáticamente respetando las propiedades de asociatividad y precedencia entre operadores, devolviendo un resultado numérico.

La evaluación de las formulas aritméticas puede formularse como un procesamiento dirigido por la sintaxis, que, a su vez, puede descomponerse modularmente en las siguientes tres tareas de procesamiento:

- Tarea de *identificación y recolección* (**idr**). Consiste en procesar el documento de entrada para registrar todas las variables declaradas en él. Implicará recorrer el árbol de derivación producido por el análisis de documento de entrada de abajo a arriba, identificando las variables y registrándolas en una tabla de variables. Tras ello, se podrá solicitar al usuario que especifique los valores para todas las variables registradas y se podrá asociar cada variable con su valor especificado en dicha tabla.

- Tarea de *propagación del entorno* (**env**). Consiste en realizar la propagación de la tabla de variables por el árbol de análisis sintáctico. Implicará recorrer el árbol de derivación asociado al documento de entrada de arriba a abajo, para permitir que la tabla de variables esté disponible y pueda ser consultada en cualquier nivel y en todo momento.

- Tarea de *evaluación* (**eva**). Consiste en procesar el documento de entrada para realizar las operaciones aritméticas a partir de los valores numéricos obtenidos del documento y de los valores de las variables aportados por la tabla de variables. Implicará recorrer el árbol de derivación de abajo a arriba, resolviendo las operaciones de suma y multiplicación de acuerdo a las reglas de asociatividad y precedencia de las mismas.

5.2.2 Establecimiento de la gramática EBNF de base

En esta actividad se abstrae la gramática documental mediante una gramática EBNF que modele las características estructurales básicas del lenguaje de marcado. Tal y como se ha descrito en el Capítulo 4, la gramática típicamente contendrá un no terminal por cada tipo de elemento, así como otros no terminales introducidos por motivos de legibilidad y/o conveniencia. Como ya se ha indicado en tal capítulo, dichos no terminales se denominan no terminales *de núcleo*. Las reglas vendrán determinadas por los modelos de contenido de los tipos de elementos. Por último, los terminales vendrán dados por las etiquetas de apertura y de cierre, y por *#pcdata* (éste denotara los contenidos textuales).

f → <f> exp </f>

exp → opnd ((a|m) opnd)*

opnd → n|v|p

a → <Add/>

m → <Mul/>

v → <Var> #pcdata </Var>

n → <Num> #pcdata </Num>

p → exp

Figura 5.2.3. Gramática EBNF de base de las fórmulas aritméticas.

En el caso de ejemplo de las fórmulas aritmética, la Figura 5.2.3 muestra una gramática EBNF de base, que mimetiza directamente la DTD de la Figura 5.2.2 introduciendo un no terminal por cada tipo elemento, así como un no terminal adicional por cada entidad tipo parámetro (<Add/> y <Mul/> son abreviaturas de <Add></Add> y <Mul></Mul> respectivamente).

5.2.3 Caracterización de la sintaxis de las vistas

Esta actividad se centra en la especificación de la sintaxis de cada vista gramatical que describirá una tarea de procesamiento del problema planteado. Dicha sintaxis vendrá caracterizada por una gramática incontextual (BNF) conforme (en el sentido introducido en el Capítulo 4) con la gramática EBNF de base. Así mismo, dicha sintaxis debe ser no ambigua, para asegurar que impone un único árbol de análisis sintáctico sobre cada documento. Dado que la ambigüedad de una gramática incontextual es una propiedad indecidible [Bar-Hillel et al. 1961], puede restringirse las especificaciones al ámbito de un tipo particular de gramáticas que no sean ambiguas (por ejemplo, gramáticas LALR(1)).

Obsérvese que, al ser cada vista gramatical independiente de las demás, es posible diseñar una sintaxis óptima y exclusiva para realizar individualmente cada tarea. Este hecho puede ejemplificarse en la tarea de evaluación de fórmulas aritméticas. En dicho ejemplo, es natural asociar una vista gramatical potencialmente diferente con cada tarea de procesamiento (*identificación y recolección* **idr**, *propagación del entorno* **env** y *evaluación* **eva**). De esta manera, las tareas **idr** y **env** no requieren de una sintaxis compleja y que capture características especiales para realizar cómputos complejos (por ejemplo, prioridades y asociatividades de operadores). Basta con utilizar una estructura simple, de carácter reconocedor, como su gramática base. Sin embargo, la tarea **eva** deberá poseer una estructura más refinada en la que se caracterice la precedencia y asociatividad de los operadores aritméticos.

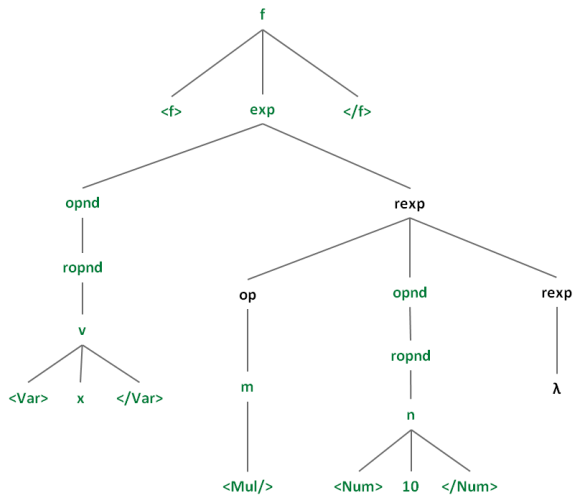
El hecho de que las sintaxis de las vistas gramaticales sean independientes entre sí no quiere decir que debamos utilizar gramáticas que no tengan nada en común entre ellas. Por el contrario, dichas sintaxis pueden compartir algunas reglas, y diferir en otras. Así, por ejemplo, como ya se ha indicado, las vistas para **idr** y para **env** comparten la misma gramática. Dicha gramática puede diferir de la asociada a **eva** en la realización de la sintaxis de las expresiones, a fin de permitir modelizar las prioridades y asociatividades de los operadores, pero coincidir con la misma en la sintaxis del resto de no terminales de núcleo. La Figura 5.2.4 muestra las sintaxis resultantes para este ejemplo.

Gramática EBNF	Vista idr y env	Vista eva
f → <f> exp </f>	f ::= <f> exp </f>	
exp → opnd ((a|m) opnd)*	exp ::= opnd rexp rexp ::= op opnd rexp rexp ::= λ op ::= a op ::= m	exp ::= expr expr ::= expr a term expr ::= term term ::= term m opnd term ::= opnd
opnd → n|v|p	opnd ::= n opnd ::= v opnd ::= p	
n → <Num> #pcdata </Num>	n ::= <Num> #pcdata </Num>	
v → <Var> #pcdata </Var>	v ::= <Var> #pcdata </Var>	
p → exp	p ::= exp	
a → <Sum/>	a ::= <Sum> </Sum>	
m → <Mul/>	m ::= <Mul> </Mul>	

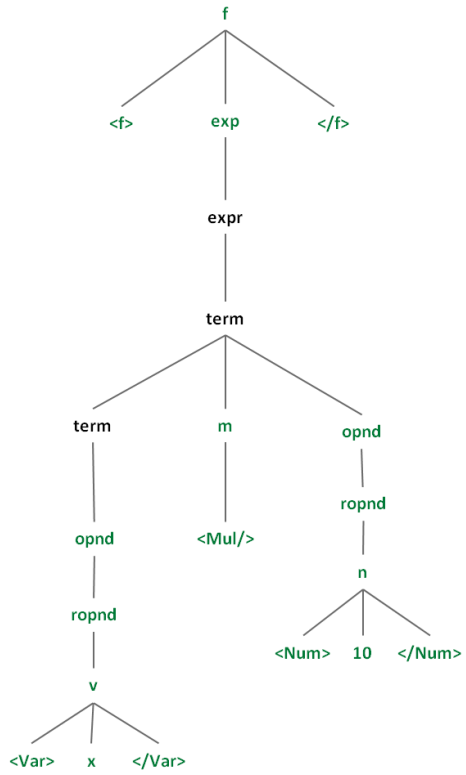
Figura 5.2.4. Estructuras gramaticales de las vistas idr, env y eva, y correspondencia con la gramática EBNF de base.

Es interesante, así mismo, remarcar que, al ser no ambiguas, cada una de las sintaxis de las vistas definidas para cada tarea de procesamiento produce un árbol de análisis sintáctico único al analizar una cadena de entrada. En el caso de ejemplo, los árboles de análisis sintáctico que se derivan de las tareas **idr** y **env** para cada documento analizado serán exactamente los mismos, al presentar ambas vistas la misma sintaxis. Sin embargo, la vista **eva** presenta un fragmento de estructura que difiere de las demás vistas. Por ejemplo, si consideramos como entrada el documento correspondiente a la expresión "x*10", las vistas **idr** y **env** producirán el árbol de análisis de la izquierda en la Figura 5.2.5, que es diferente respecto al árbol producido por la vista **eva** de la derecha en la figura anterior.

Cadena de entrada: "x*10"



(a) Árbol de análisis sintáctico de la vista idr y env.



(b) Árbol de análisis sintáctico de la vista eva.

Figura 5.2.5. Árboles de análisis sintáctico para las vistas idr, env y eva, derivados de la cadena de entrada: "x*10".

Como ya se resaltó anteriormente, una de las características importantes de las gramáticas de atributos multivista consiste en poder evaluar los árboles de análisis sintácticos derivados de las vistas de una manera eficiente, así como establecer una vía de comunicación óptima entre todas estas estructuras. Las estructuras mostradas en la Figura 5.2.5 no son aptas para tal cometido. No obstante, si se combina ambos árboles de manera inteligente, se obtendrá una representación compacta y sin duplicidad ni redundancia de nodos en el árbol. En el caso de ejemplo, los árboles de las vistas **idr** y **env**, al ser los mismos, pueden colapsar en un único

árbol para su representación. Por otra parte, el árbol derivado por la gramática de la vista **eva** tiene en común con el correspondiente a **idr**/**env** varios fragmentos (se representan en verde). Dichos fragmentos pueden compartirse. En concreto, los subárboles de **idr**/**env** y **eva** cuyos nodos raíz son **opnd** y **m** pueden compartirse. Así mismo, para que los niveles superiores del árbol también puedan compartir estructuras idénticas, es necesario detectar bajo qué nodos se producen ambigüedades. Para las tres vistas, dicha ambigüedad se produce en el nodo símbolo de núcleo **exp**, pues en la vista **eva** posee una estructura gramatical diferente a la de las vistas **idr** y **env** definidas bajo el símbolo **exp**. Permitiendo que este nodo sea especial y pueda contener varios subárboles diferentes, se consigue mantener la compartición de nodos a niveles superiores. Esta nueva representación es ahora una estructura de bosque, que se muestra en la Figura 5.2.6. La estructura en sí es la utilizada en los algoritmos de análisis sintáctico ascendente generalizados (sección 5.3). De hecho, esta combinación inteligente de los distintos árboles de derivación correspondientes con las diferentes sintaxis de las vistas, en una única estructura de bosque, tiene como ventaja, además de representar de manera eficiente todos los árboles de derivación en una misma estructura, la posibilidad de ser construido mediante un analizador sintáctico de carácter general, tal y como se detallará en la sección 5.3.

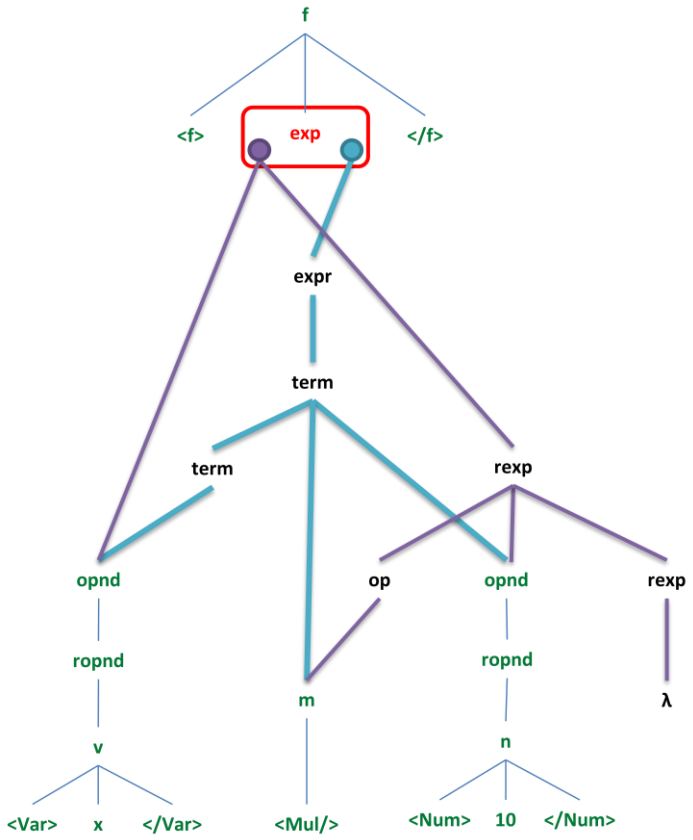


Figura 5.2.6. Bosque de análisis sintáctico que resulta de combinar los árboles de la Figura 5.2.5.

5.2.4 Caracterización de la semántica de las vistas

Una vez elaboradas las sintaxis de las vistas gramaticales para cada tarea de procesamiento, es necesario extenderlas con atributos y ecuaciones semánticas para definir cómo se realizan los cómputos de dichas tareas. Los atributos se consideran locales a cada vista. De esta forma, aunque dos vistas diferentes pueden introducir el mismo nombre de atributo para un mismo símbolo gramatical, no se tratará de un único atributo, sino de un atributo diferente para cada vista. Por lo demás, las extensiones semánticas cumplen con las restricciones usuales de las gramáticas de atributos, con los siguientes añadidos:

- En las reglas de una determinada vista será posible referir a atributos sintetizados definidos en otras vistas de los símbolos de núcleo que aparecen en los cuerpos de dichas reglas.

- Así mismo, en las reglas de las vistas para los símbolos de núcleo (es decir, que tengan símbolos de núcleo como sus cabezas) será posible referir a atributos heredados de dichos símbolos definidos en otras vistas.

En las descripciones, la referencia a dichos atributos externos, definidos en otras vistas, se hará indicando el nombre de la vista entre corchetes junto al correspondiente atributo. Estos dos añadidos (la posibilidad de utilizar atributos sintetizados en otras vistas en los símbolos de núcleo de los cuerpos de las reglas, así como atributos heredados externos cuando dichos símbolos ocurren en la cabeza) constituyen el mecanismo básico de comunicación entre las diferentes tareas de procesamiento. Efectivamente, a través de estos atributos se permite la transmisión de datos entre distintas tareas de procesamiento correspondientes a diferentes vistas. Así, por ejemplo, la tarea modelizada por una vista V_1 puede proporcionar una entrada a la tarea modelizada por una vista V_2 mediante un determinado atributo heredado de un símbolo de núcleo. Dicha entrada puede, entonces, elaborarse en V_2, ofreciendo como resultado el valor de un atributo sintetizado del mismo o de otro símbolo de núcleo, valor que puede utilizarse como salida en V_1.

Por último, como ya se introdujo en el Capítulo 3, el terminal *#pcdata* posee sólo un atributo por defecto denominado *text*. Dicho atributo aloja el contenido textual del documento XML designado por *#pcdata*. Las etiquetas de cierre no poseen atributos. En las etiquetas de apertura, sus nombres de atributos se corresponden con los nombres de atributos respectivos definidos en el documento XML para dicha etiqueta.

Vista env	Vista idr	Vista eva
f ::= <f> exp </f> exp.eh = mkenv(**exp.ids**)	*f ::= <f> exp </f>*	**f ::= <f> exp </f>** f.v = printResult(exp.v)
exp ::= opnd rexp opnd.eh = exp.eh rexp.eh = exp.eh **rexp ::= op opnd rexp** opnd.eh = rexp(0).eh rexp(1).eh = rexp(0).eh **rexp ::= λ** **op ::= a** **op ::= m**	**exp ::= opnd rexp {** exp.ids = opnd.ids ∪ rexp.ids **rexp ::= op opnd rexp {** rexp(0).ids = opnd.ids ∪ rexp(1).ids **rexp ::= λ** rexp.ids = ∅ **op ::= a** **op ::= m**	**exp ::= expr** exp.v = expr.v **expr ::= expr a term** expr(0).v = expr(1).v + term.v **expr ::= term** expr.v = term.v **term ::= term m opnd** term(0).v = term(1).v * opnd.v **term ::= opnd** term.v = opnd.v
opnd ::= n **opnd ::= v** **opnd ::= p** p.eh = opnd.eh	**opnd ::= n** opnd.ids = ∅ **opnd ::= v** opnd.ids = {v.id} **opnd ::= p** opnd.ids = p.ids	**opnd ::= n** opnd.v = val(n.v) **opnd ::= v** opnd.v = valueOf(**v.id opnd.eh**) **opnd ::= p** opnd.v = p.v

n ::= <Num> #pcdata </Num>	*n ::= <Num> #pcdata </Num>*	**n ::= <Num> #pcdata </Num>**
		n.v = #pcdata.text

v ::= <Var> #pcdata </Var>	**v ::= <Var> #pcdata </Var>**
	v.id = #pcdata.text
		v ::= <Var> #pcdata </Var>

**p ::= exp ** exp.eh = p.eh	**p ::= exp ** p.ids = exp.ids	**p ::= exp ** p.v = exp.v
a ::= <Sum/>	*a ::= <Sum/>*	*a ::= <Sum/>*
m ::= <Mul/>	*m ::= <Mul/>*	*m ::= <Mul/>*

Figura 5.2.7. Gramática de atributos multivista. Vistas gramaticales: propagación del entorno (rojo), identificación y recolección (naranja) y cálculo del valor (azul).

A modo de ejemplo, la Figura 5.2.7 muestra el resultado de añadir semántica a las sintaxis definidas en la Figura 5.2.4, completando, de esta forma, las vistas gramaticales para las tareas **idr**, **env** y **eva**. En dicha especificación, las ecuaciones semánticas con un atributo **subrayado en negrita**/resaltado indican el intercambio de información (y el sentido) entre tareas de procesamiento de vistas gramaticales diferentes, mediante la referencia a atributos de otras vistas. La función *mkenv* crea y devuelve la tabla de variables a partir del conjunto de nombres de variables y realiza las peticiones al usuario para que establezca el valor de dichas variables, mientras que la función *printResult* tiene como efecto lateral imprimir el resultado. De esta forma:

- La vista sintáctica **idr** fue diseñada para realizar la tarea de identificación y recolección de variables. Para conseguir dicho objetivo, se recolectará en un conjunto los nombres de las variables, explorando el árbol de derivación desde los nodos hoja hacia el nodo raíz del árbol. Este proceso puede modelizarse declarativamente asociando un atributo sintetizado **ids** con los no terminales **exp**, **opnd** y **p**. El valor de dicho atributo en los correspondientes nodos de los árboles de derivación será el conjunto de nombres de variables aludido. Dichos nombres se

obtendrán extrayendo el contenido textual del terminal *#pcdata* disponible entre las etiquetas <Var> y </Var>, como valor del atributo sintetizado **id** de **v**. De esta manera, a través del atributo sintetizado **ids** del símbolo **exp**, se dispondrá del conjunto formado por todos los nombres de variables distintos presentes en la expresión.

- La vista sintáctica **env** fue diseñada para realizar la tarea de propagación del entorno. Para conseguir dicho objetivo, se construirá una tabla de variables a partir del conjunto de nombres de variables aportado por el atributo **ids** del símbolo **exp** de la vista **idr**. La información de los valores de las variables, de la tabla de variables, será establecido por el usuario en tiempo de ejecución. Ambos aspectos se realizarán a nivel del símbolo **f**. El aspecto de propagación del entorno en sí, propagará la tabla de variables una vez rellena, por el árbol de derivación desde la raíz hacia los nodos hoja del árbol, a través de los atributos heredados **eh** de los símbolos **exp**, **opnd** y **p**.

- La vista sintáctica **eva** fue diseñada para realizar la tarea de evaluación de las fórmulas aritméticas. Para conseguir dicho objetivo, se recolectan los valores de los operandos y se aplican los operadores suma o multiplicación en el orden apropiado. El aspecto de recolección de valores de operandos numéricos, se realizará extrayendo el contenido textual del terminal *#pcdata* disponible entre las etiquetas <Num> y </Num>, y fijando con dicho valor el atributo sintetizado **v** de **n**. El aspecto de obtención del valor de un operando, se realizará fijando el valor del atributo sintetizado **v** de **opnd**. El valor cadena de un número se transforma en un valor numérico, y el valor de una variable se obtiene de la tabla de variables procedente de la vista **env** que estará disponible en el atributo heredado **eh** (definido en la vista **env**) de **opnd**. El valor de una subfórmula no necesitará conversión alguna, pues el valor de **v** para **p** será directamente el valor numérico de resolver tal subfórmula directamente a través de su atributo sintetizado **v**. Por último, el aspecto de evaluar una suma o una multiplicación será realizado a nivel de **exp**, devolviendo en su atributo sintetizado **v** el resultado de la operación. La elección de una estructura sintáctica apropiada tendrá en cuenta, en este caso, los aspectos de precedencia y asociatividad de operadores, aspectos que son meramente estructurales.

Es interesante, por último, examinar el efecto de una especificación como gramática de atributos multivista sobre las entradas procesadas. Como ya se ha visto en la sección 5.2.3, cada vista sintáctica induce un árbol de análisis sintáctico. Los árboles de análisis sintáctico de una gramática multivista pueden unirse formando una estructura de bosque. Por consiguiente, una vez que se añade semántica, este hecho redunda en una atribución de dicho bosque. La Figura 5.2.8 muestra el resultado de atribuir el bosque en la Figura 5.2.6 junto con el grafo de dependencias entre los atributos de dicho bosque.

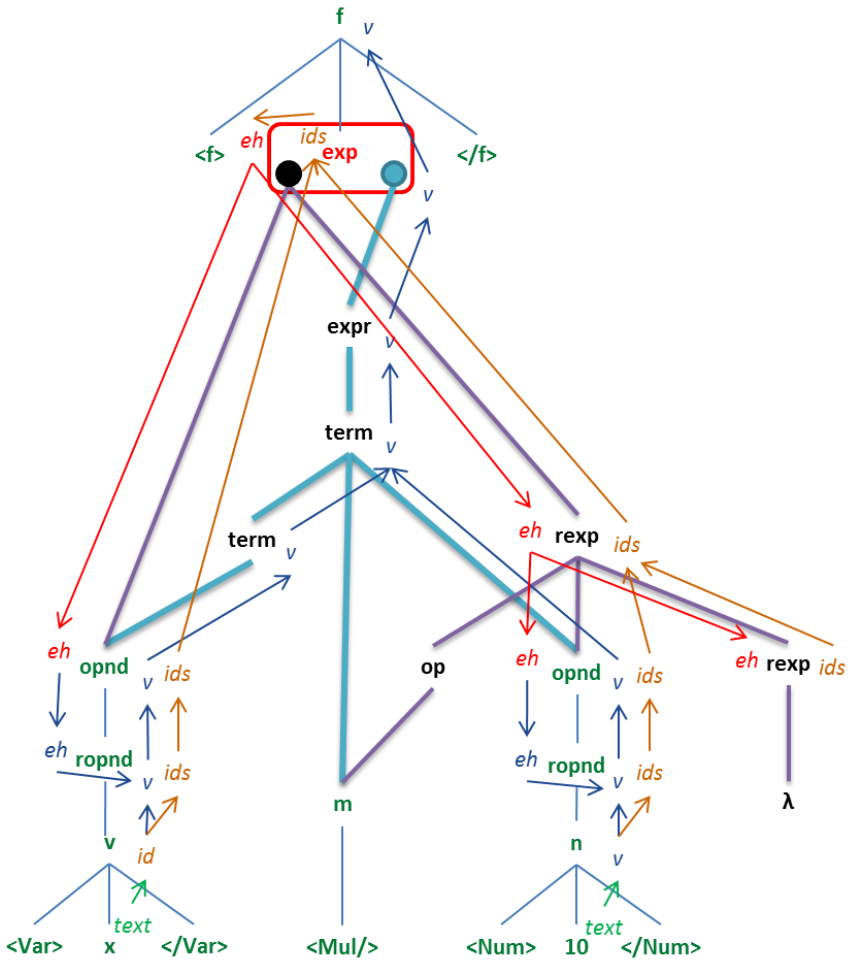


Figura 5.2.8. Atribución y grafo de dependencias para el bosque de análisis sintáctico de las vistas idr (naranja), env (rojo) y eva (azul).

5.2.5 Ejecución de la especificación

El modelo de ejecución de una gramática de atributos multivista es análogo al modelo de ejecución de una gramática de atributos convencional, aunque especializado para las peculiaridades de la nueva propuesta. De esta forma, dicho modelo pude estructurarse en dos fases bien diferenciadas:

- *Fase de análisis*. En dicha fase el documento de entrada se analiza para obtener una representación del mismo en forma de bosque atribuido.
- *Fase de evaluación*. En dicha fase el bosque atribuido se evalúa para determinar el valor de los atributos en sus nodos.

Las siguientes secciones en este capítulo desarrollan los detalles de cada una de estas fases.

5.3 La fase de análisis

La fase de análisis consiste en el análisis del documento XML de entrada, y deberá producir como salida un bosque de análisis sintáctico con una estructura adecuada a la especificación gramatical multivista. Para llevar a cabo dicha fase proponemos dos métodos alternativos:

- Un método basado en el análisis LR simultáneo y sincronizado del documento de entrada con respecto a cada una de las vistas. Dicho método se denominará MVLR (*MultiView* LR).
- Un método equivalente, basado en los algoritmos de análisis LR generalizados, algoritmos que generalizan el método LR para tratar con clases arbitrarias de gramáticas (incluso con gramáticas ambiguas). El método resultante se denominará MVGLR (*MultiView* GLR).

La sección 5.3.1 describe el método MVLR. La sección 5.3.2 describe el método MVGLR. La sección 5.3.3 muestra, por último, cómo utilizar de manera efectiva dichos métodos para orquestar la fase de análisis de documentos XML de acuerdo con gramáticas de atributos multivista.

5.3.1 El método de análisis MVLR

Dado que las estructuras sintácticas de una gramática multivista deben haber superado un test efectivo de no ambigüedad (p.e., basado en el empleo de algún método LR), es natural plantear la fase de análisis como el análisis del documento respecto a cada una de las vistas (o, más precisamente, respecto a cada una de las estructuras sintácticas diferentes, ya que pueden existir vistas que compartan exactamente la misma estructura). Para evitar repetir el análisis para cada vista, es, además, posible realizar dichos análisis de forma simultánea, y sincronizada. En esta sección desarrollamos esta idea. Como resultado proponemos un primer método de análisis (el anteriormente mencionado método MVLR). Dado que dicho método integra simultáneamente *n* análisis LR, la sección 5.3.1.1 detalla el método de análisis LR ya introducido en el Capítulo 2. La sección 5.3.1.2 desarrolla el método MVLR en sí.

5.3.1.1 El algoritmo LR

Como ya se ha comentado en el Capítulo 2, el método de análisis LR es un método de análisis ascendente, guiado por un autómata reconocedor de prefijos viables (las posibles configuraciones de la pila del analizador) [Knuth 1965]. Dicho autómata se codifica en un par de tablas, una para no terminales (tabla IR_A), y otra para terminales (ACCION). Esta última contiene, además, información sobre las acciones a realizar por el analizador (reducir, desplazar, aceptar).

```
PARSE(S0,ACCION,IR_A):
  π = [S0]
  repetir
    caso ACCION(head(π),Simbolo(TokenActual()))
      shift S: N = nuevoNodoTerminal(TokenActual())
        push(π,N)
        push(π,S)
      reduce A ::= α:
        <S',τ> = POPBODY(Π,α)
        N = nuevoNodoNoTerminal(A, τ)
        push(π,N)
        push(π, IR_A(S',A))
      accept: devuelve OK
      en otro caso: devuelve ERROR
    fin caso
    SiguienteToken()
  fin repetir
```

Figura 5.3.1. Pseudocódigo para el algoritmo LR.

El pseudocódigo de la Figura 5.3.1 detalla el algoritmo como un procedimiento *PARSE*, que, además de realizar el reconocimiento de la entrada, construye el árbol de análisis sintáctico para la misma. Dicho procedimiento toma como entradas: (i) el estado inicial del autómata LR (S0), y (ii) las tablas de análisis ACCION e IR_A. Por lo demás, la lógica del algoritmo es simple, y refleja fielmente el funcionamiento de este tipo de analizadores. Efectivamente, el algoritmo comienza inicializando la pila de análisis (π) con el estado inicial. Seguidamente opera iterativamente, determinando en cada iteración la acción a ejecutar, en base al estado de la cima de la pila, y ejecutando dicha acción:

- La ejecución de una acción de desplazamiento supone apilar una referencia al nodo del árbol de análisis sintáctico asociado con el token actual (obsérvese que, en lugar de apilar símbolos, es posible apilar directamente referencias a nodos del árbol). Seguidamente se apila el estado al que se transita.

- La ejecución de una acción de reducción por $A ::= \alpha$ supone, primeramente, desapilar $2|\alpha|$ símbolos de la pila de análisis (para cada símbolo, el estado al que se transita por dicho símbolo, y la referencia al correspondiente nodo en el árbol de análisis sintáctico). Así mismo, durante este desapilado se acumula la secuencia de nodos hijo. Este comportamiento se plasma en la función *POPBODY* de la Figura 5.3.2. Una vez hecho esto, se aplica la tabla IR_A sobre el estado que resulta en la cima de la pila, y sobre *A*, para determinar el estado siguiente. Así mismo, se construye un nuevo nodo en el árbol asociado con *A*, y que tiene como hijos los acumulados en el proceso de desapilado anterior.

- La ejecución de una acción de aceptación supone finalizar con éxito el proceso.

- Por último, la ausencia de acción supone el rechazo como errónea de la entrada.

De esta forma, el algoritmo proporciona una alternativa adecuada para el análisis de un documento con respecto a cada una de las estructuras sintácticas, y es, tal y como se ha hecho patente en el Capítulo 3, la solución de análisis habitualmente adoptada en nuestros trabajos previos sobre desarrollo dirigido por lenguajes de aplicaciones de procesamiento XML [Sarasa et al. 2009-a, Sarasa et al. 2009-b]. No obstante, dado que, en una gramática multivista, se maneja más de una gramática incontextual, es necesario generalizar el método para tratar con múltiples gramáticas.

```
POPBODY (Π,α):
  τ =  []
  para i=1 hasta |α|
     pop(Π)
     N = pop(Π)
     push(τ,N)
  fin para
  devuelve <head(Π),τ>
```

Figura 5.3.2. Pseudocódigo para la función POPBODY.

5.3.1.2 El algoritmo MVLR

La idea básica para orquestar la fase de análisis con el método LR es, como ya se ha indicado, desarrollar simultáneamente *n* análisis LR del documento de entrada, uno por cada estructura sintáctica diferente. Para ello:

- Será necesario sincronizar dichos análisis en los desplazamientos, y también en las reducciones correspondientes a los no terminales de núcleo (*reducciones de núcleo*), ya que, en dichas reducciones, será necesario empaquetar los subárboles correspondientes a las diferentes gramáticas en nodos con más de una lista de hijos.

- Dado que los terminales de núcleo pueden exhibir una estructura común en más de una subgramática, para evitar redundancias en el proceso de construcción del bosque de análisis será necesario asignar previamente cada subgramática de cada no terminal de núcleo a una única gramática, que se denominará la *gramática constructora* de dicha subgramática (y, por ende, de todas las producciones de la misma). La asignación puede realizarse de manera arbitraria, ya que, en último término, la estructura resultante se integrará en el nodo correspondiente al no terminal de núcleo, que será compartido por los procesos de análisis de todas las gramáticas que intervienen en el análisis. Dicho preproceso previo puede resumirse en una función *informativa sobre la construcción* Θ que, dado un número de

gramática *i* y una producción *A* ::= α, permite determinar si la gramática *i* es (Θ(*i*,*A*::= α) *cierto*) o no (Θ(*i*,*A* ::= α) *falso*) la gramática constructora para *A* ::= α.

```
PARSE(NGs, S0s, ACCIONes, IR_As, Θ, N):
  para i=1 hasta NGs
    Π[i] = [S0s[i]]
  fin para
  repetir
    repetir
      REDUCENONUCLEO(Π,NGs,ACCIONes,IR_As, Θ,N)
      reduccionesRealizadas = REDUCENUCLEO(Π,NGs,ACCIONes,IR_As, Θ)
    hasta no reduccionesRealizadas
    si ACCIONes(1,head(Π[1]),Simbolo(TokenActual())) = accept
      devuelve OK
    fin si
    si ACCIONes(1,head(Π[1]),Simbolo(TokenActual())) = ⊥
      devuelve ERROR
    fin si
    DESPLAZA(Π,NGs)
    SiguienteToken()
  fin repetir
```

Figura 5.3.3. Pseudocódigo para el algoritmo MVLR (procedimiento PARSE).

El pseudocódigo de la Figura 5.3.3 detalla la estrategia, de nuevo como un procedimiento *PARSE*. Dicho procedimiento:

- Toma como parámetros: (i) el número de gramáticas que intervienen en el análisis (NGs), (ii) los estados iniciales de cada gramática s (función S0s: dado un número de gramática *i*, S0s(*i*) representa el estado inicial de dicha gramática), (iii) las tablas de análisis de cada gramática (ACCIONes e IR_As; así, por ejemplo, dado un estado de análisis *s*, un no terminal *a*, y un número de gramática *i*, mediante ACCIONes(*i*,*s*,*a*) se accederá a la acción que, según el autómata LR correspondiente con la gramática *i*, debe llevarse a cabo cuando, en el estado *s*, se observa *a* como siguiente terminal), (iv) la función informativa sobre la construcción Θ, y (v) una función *informativa sobre núcleos* N que determina si un no terminal *A* es (N(*A*) *cierto*), o no (N(*A*) *falso*) un símbolo de núcleo.

- Plasma la estrategia sugerida anteriormente. Para ello, comienza inicializando, para cada gramática, la correspondiente pila de análisis al estado inicial de dicha gramática (Π[i] denota la pila asociada a la gramática *i*-ésima). Seguidamente itera una *fase de reducción* seguida de una *comprobación de aceptación*, seguida de una *comprobación de error*, seguido una *fase de desplazamiento*. Por último, consume el token actual. La iteración finaliza, bien porque se decide la aceptación de la entrada, bien porque la entrada se rechaza como errónea.

```
REDUCENONUCLEO(Π,NGs,ACCIONes,IR_As,Θ,N):
  para i=1 hasta NGs
    mientras ACCIONes(i,head(Π[i]),Simbolo(TokenActual())) = reduce A ::= α y no N(A)
      <S',τ> = POPBODY(Π[i],α)
      si Θ(i,A::=α)
        N = nuevoNodoNoTerminal(A, τ)
      en otro caso
        N = ⊥
      fin si
      push(Π[i],N)
      push(Π[i], IR_A(S',A))
    fin mientras
  fin para
```

Figura 5.3.4. Pseudocódigo para el algoritmo MVLR (procedimiento REDUCENONUCLEO).

La *fase de reducción* implica, a su vez, la iteración, hasta que no puedan realizarse más reducciones, de las dos subfases siguientes:

- Realización de todas las *reducciones no de núcleo* (Figura 5.3.4). Estas son reducciones de la forma *A* ::= α con *A* un no terminal *no* de núcleo. Básicamente esta subfase consiste en aplicar acciones de reducción no de núcleo sobre cada pila de análisis, hasta que se alcance un punto en el que no es posible aplicar más acciones de este tipo. La forma de aplicar cada reducción es análoga a la que se sigue en el método LR discutido en la sección anterior. Obsérvese, así mismo, que cada reducción implica operaciones de construcción en el bosque de análisis sintáctico únicamente para producciones que son constructoras en la correspondiente gramática. En este caso, la construcción es análoga a la realizada por las reducciones del método LR: crear un nodo para la cabeza de la producción, y fijar sus nodos hijo a los acumulados durante el proceso de desapilado integrado en la operación de reducción.

- Realización de las *reducciones de núcleo* (Figura 5.3.5). Una vez finalizada la subfase anterior, puede ocurrir que, bien en la primera de las pilas sea aplicable una acción de reducción, bien sea aplicable otro tipo de acción (o incluso no sea aplicable ninguna, en caso de error). En el primer caso, dicha reducción ha de serlo por una producción de la forma *A* ::= α, con *A* un no terminal de núcleo (es decir, dicha reducción será *de núcleo*). En este caso, y dado que todas las gramáticas deben ser conformes con la gramática EBNF de base (y, por tanto, conformes entre sí), debe ocurrir que en *todas* las otras pilas sea aplicable también una reducción de núcleo para *A*. En esta subfase se aplican, por tanto, estas reducciones en *cada una* de las pilas. Así mismo, se crea un nodo para el símbolo *A*, en el que se registran sucesivamente, como listas de nodos hijo alternativas, la secuencia de hijos obtenidas por cada reducción (se excluyen aquellas reducciones por producciones

no constructoras en las respectivas gramáticas). La referencia a dicho nodo se aloja, entonces, en cada una de las pilas al realizar la correspondiente transición, lo que permite a su vez, continuar la construcción del bosque de análisis.

```
REDUCENUCLEO(Π,NGs,ACCIONes,IR_As,Θ):
  si ACCIONes(1,Head(Π[1]),Simbolo(TokenActual())) = reduce A ::= α
    N = nuevoNodoNoTerminal(A)
    para i=1 hasta NGs
      reduce A ::= β =  ACCIONes(i,head(Π[i]),Simbolo(TokenActual()))
      <S',τ> = POPBODY(Π[i], β)
      si Θ (i,A::= β)
        añadeHijos(N, τ)
      fin si
      push(Π[i],N)
      push(Π[i], IR_A(S',A))
    fin para
    devuelve cierto
  fin si
  devuelve falso
```

Figura 5.3.5. Pseudocódigo para el algoritmo MVLR (procedimiento REDUCENUCLEO).

Por su parte, la fase de *desplazamiento* (Figura 5.3.6) sigue una estrategia análoga a la seguida en la subfase de realización de las reducciones de núcleo: se construye un nodo para el terminal, se desplaza en cada una de las pilas una referencia al mismo, y se transita al estado correspondiente. Obsérvese que, de nuevo, y por la restricción de conformidad, el simple hecho de que en la primera de las pilas pueda llevarse a cabo un desplazamiento, implica que dicho desplazamiento podrá llevarse a cabo en todas las otras pilas.

```
DESPLAZA(Π,NGs):
  N = nuevoNodoTerminal(TokenActual())
  para i=1 hasta NGs
    shift S = ACCIONes(i,head(Π[i]),Simbolo(TokenActual()))
    push(Π,N)
    push(Π,S)
  fin para
```

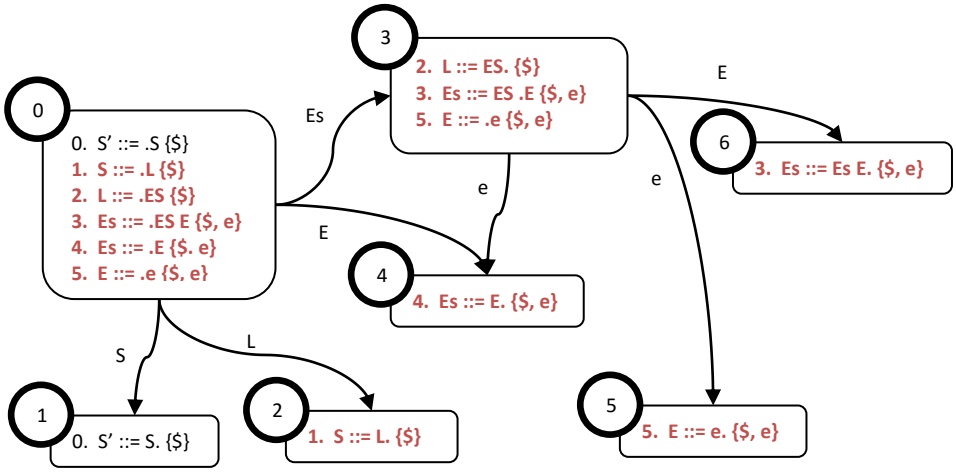
Figura 5.3.6. Pseudocódigo para el algoritmo MVLR (procedimiento DESPLAZA).

En lo referente a la *comprobación de aceptación*, se comprueba que en la primera de las pilas sea aplicable la acción de aceptación, y, si es así, se termina el análisis con éxito (se supone, entonces, que el bosque de análisis podrá obtenerse de un componente de construcción adecuado, que ofrezca los servicios de construcción que aparecen en los pseudocódigos). Obsérvese que, de nuevo, por la restricción de conformidad y por el hecho de que todas las reducciones han sido ya agotadas, dicha acción será también aplicable en el resto

de las pilas, ya que, si no, bien alguna de las pilas sería indicativa de error, bien alguna de las pilas obligaría a desplazar: ambos supuestos contradicen la restricción de conformidad.

Por último, la *comprobación de error* comprueba que en la primera de las pilas no sea aplicable ninguna acción (la *no acción* se representa con ⊥ en el pseudocódigo).

Autómata LALR(1) para la vista 1



Autómata LALR(1) para la vista 2

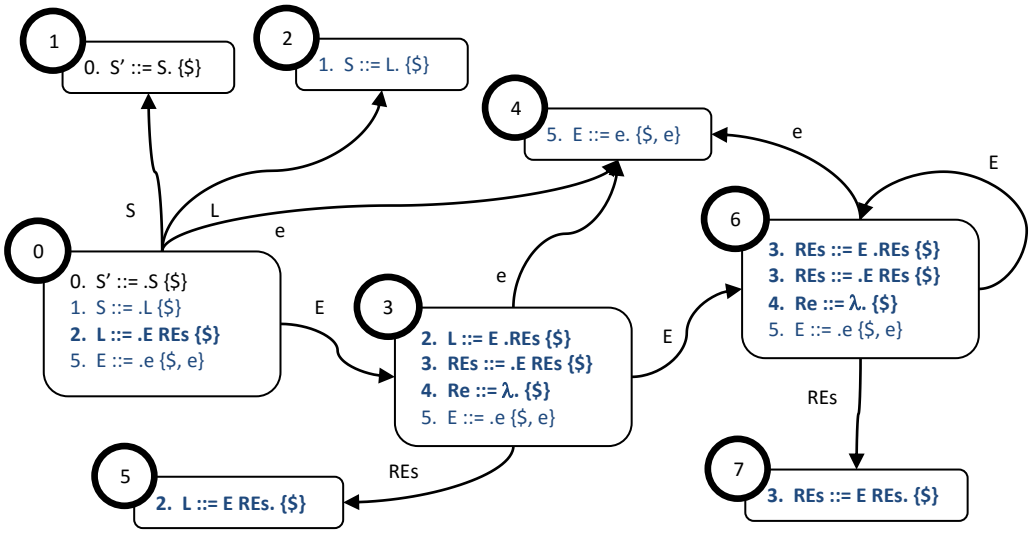


Figura 5.3.7. Autómatas LALR(1) para la vista 1 y vista 2.

Gramática vista 1	Gramática vista 2
1. S ::= L	1. S ::= L
2. L ::= Es **3. Es ::= Es E** **4. Es ::= E**	**2. L ::= E REs** **3. REs ::= E REs** **4. REs ::= λ**
5. E ::= e	5. E ::= e

Figura 5.3.8. Estructura multivista de ejemplo.

Para ilustrar el comportamiento de este algoritmo se considerará la estructura multivista mostrada en la Figura 5.3.8. En dicha figura se asocia también un identificador único a cada producción (situado a su izquierda), y se indica la vista constructora para la misma (con sus producciones en **negrita**).

Tabla de análisis sintáctico LALR(1) para la vista 1

Reglas Gramaticales:	**1. S ::= L** **2. L ::= Es** **3. Es ::= Es E** **4. Es ::= E** **5. E ::= e**					
Estado	**e**	**$**	**S**	**L**	**Es**	**E**
0	Shift 5		1	2	3	4
1		Accept				
2		Reduce 1				
3	Shift 5	Reduce 2				6
4	Reduce 4	Reduce 4				
5	Reduce 5	Reduce 5				
6	Reduce 3	Reduce 3				

Tabla de análisis sintáctico LALR(1) para la vista 2

Reglas Gramaticales:	1. S ::= L **2. L ::= E REs** **3. REs ::= E REs** **4. REs ::= λ** 5. E ::= e					
Estado	**e**	**$**	**S**	**L**	**REs**	**E**
0	Shift 4		1	2		3
1		Accept				
2		Reduce 1				
3	Shift 4	Reduce 4			5	6
4	Reduce 5	Reduce 5				
5		Reduce 2				
6	Shift 4	Reduce 4			7	6
7		Reduce 3				

Figura 5.3.9. Tabla de análisis sintáctico LALR(1) para la vista 1 y la vista 2.

La Figura 5.3.7 muestra el autómata LALR(1) para cada una de las gramáticas (vista 1 y vista 2). Las respectivas tablas de análisis LALR(1) se muestran en la Figura 5.3.9. Las Figura 5.3.10 y Figura 5.3.11 muestran, por último, la evolución de las pilas, las acciones realizadas, y el estado de construcción del bosque de análisis sintáctico para el análisis de la entrada "ee".

Entrada	Pila vista 1	Acción vista 1	SPPF	Pila vista 2	Acción vista 2
e	0	(0, d1): Shift 5	d1: **e**	0	(0, d1): Shift 4
e	5 d1 0	(5, n1): Reduce 5 "E ::= e" (0, **E**): 4	n1: **E** d1: **e**	4 d1 0	(4, n1): Reduce 5 "E ::= e" (0, **E**): 3
	4 n2 0	(4, e1): Reduce 4 "Es ::= E" (0, Es): 3	e1: **Es** n1: **E** d1: **e**		
	3 e1 0	(3, d2): Shift 5	e1: **Es** n1: **E** d1: **e** d2: **e**	3 d1 0	(3, d2): Shift 4
$	5 d2 3 e1 0	(5, n2): Reduce 5 "E ::= e" (3, **E**): 6	e1: **Es** n1: **E** n2: **E** d1: **e** d2: **e**	4 d2 3 d1 0	(4, n2): Reduce 5 "E ::= e" (3, **E**): 6
	6 n2 3 e1 0	(6, e2): Reduce 3 "Es ::= Es E" (0, Es): 3	e2: **Es** e1: **Es** u1: **REs** n1: **E** n2: **E** d1: **e** d2: **e**	6 n2 3 d1 0	(6, u1): Reduce 4 "REs ::= λ" (6, REs): 7
			e2: **Es** u2: **REs** e1: **Es** u1: **REs** n1: **E** n2: **E** d1: **e** d2: **e**	7 u1 6 n2 3 d1 0	(7, u2): Reduce 3 "REs ::= E REs" (3, REs): 5

Figura 5.3.10. Proceso de análisis y construcción de bosques de MVLR (parte 1).

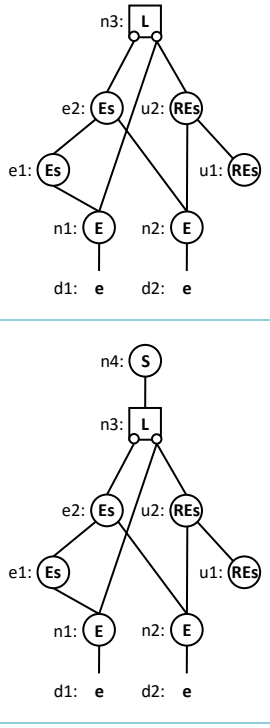
<table>
<tr><th>Entrada</th><th>Pila vista 1</th><th>Acción vista 1</th><th>SPPF</th><th>Pila vista 2</th><th>Acción vista 2</th></tr>
<tr><td>$</td><td>3
e2
0</td><td>(3, n3): Reduce 2
“L ::= Es”
(0, L): 2</td><td rowspan="3"> 
</td><td>5
u2
3
d1
0</td><td>(5, n3): Reduce 2
“L ::= E REs”
(0, L): 2</td></tr>
<tr><td>2
n3
0</td><td>(2, n4): Reduce 1
“S ::= L”
(0, S): 1</td><td>2
n3
0</td><td>(2, n4): Reduce 1
“S ::= L”
(0, S): 1</td></tr>
<tr><td>1
n4
0</td><td>(1): Accept</td><td>1
n4
0</td><td>(1): Accept</td></tr>
</table>

Figura 5.3.11. Proceso de análisis y construcción de bosques de MVLR (parte 2).

5.3.2 El método de análisis MVGLR

Una de las principales deficiencias del método MVLR es que los *n* procesos de análisis simultáneos que integra pueden realizar análisis redundantes de porciones del documento con respecto a las subgramáticas comunes a las gramáticas involucradas. Para solventar este problema, en esta tesis se ha explorado el uso de algoritmos de análisis generales, capaces de tratar gramáticas generales, incluso ambiguas, mediante la exploración sistemática y eficiente de las distintas alternativas de análisis. Como resultado, se ha llevado a cabo una reelaboración del método MVLR basada en este tipo de algoritmos (más concretamente, basada en el algoritmo GLR, *Generalized Left to Right*, original propuesto por el Informático y Biólogo Molecular japonés Masaru Tomita [Tomita 1986]). La sección 5.3.2.1 describe la propuesta original de Tomita del algoritmo GLR. La sección 5.3.2.2 discute las deficiencias que dicho algoritmo presenta de cara a su aplicación directa a las gramáticas multivista, y algunas heurísticas para paliar dichas deficiencias. La sección 5.3.2.3 describe, por último, el método propuesto.

5.3.2.1 El algoritmo GLR de Tomita

A diferencia de los algoritmos de análisis sintáctico específicos, como los ascendentes y descendentes aludidos en el Capítulo 2, que se especializan en el manejo eficiente de una determinada clase de gramáticas incontextuales de tipo LR(k) [Knuth 1965] o LL(k) [Aho et al. 2006], los algoritmos generales como [Earley 1968] o [Younger 1967] se enfrentan al manejo de cualquier clase de gramática incontextual a costa de una mayor complejidad en el procesamiento. A diferencia de las gramáticas LR(k) o LL(k), las gramáticas generales pueden presentar ambigüedad o requerir patrones de predicción mucho más sofisticados. El método GLR (Generalized Left to Right) [Tomita 1986], se presenta, de esta forma, como el algoritmo más eficiente capaz de manejar gramáticas incontextuales de carácter general. Su modo de funcionamiento es muy similar al de un analizador sintáctico ascendente de desplazamiento-reducción. Es más, se guía por el mismo tipo de tablas de análisis sintáctico construidas para un analizador ascendente a las que se ha hecho alusión anteriormente, aunque, en este caso, admite celdas con múltiples entradas. A modo de ejemplo, la Figura 5.3.12 muestra una gramática ambigua, que resulta de unir las dos vistas en la Figura 5.3.8. Las tablas de análisis que se generan por el método LALR(1) (Figura 5.3.12) introducen, por tanto, celdas con múltiples acciones aplicables (conflictos).

Gramática ambigua:	1. S ::= L 2. L ::= Es 3. Es ::= Es E 4. Es ::= E 5. L ::= E REs 6. REs ::= E REs 7. REs ::= λ 8. E ::= e						
Estado	**e**	**$**	**S**	**L**	**Es**	**REs**	**E**
0 (inicial)	Shift 5		1	2	3		4
1		Accept					
2		Reduce 1					
3	Shift 5	Reduce 2					6
4	Reduce 4 Shift 5	Reduce 4 Reduce 7				7	8
5	Reduce 8	Reduce 8					
6	Reduce 3	Reduce 3					
7		Reduce 5					
8	Shift 5	Reduce 7				9	8
9		Reduce 6					

Figura 5.3.12. Gramática ambigua de unificar las gramáticas vista 1 y vista 2 de la Figura 5.3.8, y su tabla de análisis sintáctico LALR(1) (presenta conflictos en sus celdas).

Siguiendo el modelo de operación de un analizador ascendente por desplazamiento-reducción, para poder tratar los conflictos reflejados por las entradas múltiples en las celdas de la tabla de análisis sintáctico se requerirá bifurcar, idealmente, la pila de análisis sintáctico en varias pilas ante la presencia de conflictos, a fin de explorar sistemáticamente todas las posibilidades. De esta forma, el algoritmo GLR utiliza una estructura de datos eficiente, denominada G*raph* S*tructured* S*tack* (GSS), para gestionar estas bifurcaciones. La GSS es un grafo dirigido cuyas bifurcaciones son las pilas de análisis sintáctico adicionales que se requieren para tratar los conflictos, una por cada alternativa. No obstante, aparte de contener bifurcaciones, la estructura también contiene reunificaciones, cuando, por distintos caminos, se llega a estados de análisis equivalentes. Esta estructura permite al algoritmo GLR, por tanto, realizar un proceso eficiente con un coste en espacio reducido. Más precisamente, el GSS se compone de:

- *Nodos,* que representan estados del autómata LR utilizados para construir las tablas de análisis.
- Arcos, etiquetados con referencias a nodos en el bosque de análisis sintáctico.

Por su parte, el bosque de análisis sintáctico se representa mediante una estructura denominada *Shared-Packed Parse Forest* (SPPF), cuyos nodos interiores contienen cero o más listas de nodos hijo.

El método parte de una GSS con un único nodo, correspondiente al estado inicial del autómata LR utilizado para construir la tabla de análisis, y termina reconociendo la entrada cuando existe una cima en la GSS correspondiente a un nodo que tiene asociado una acción de aceptación. El modo de operación del algoritmo GLR se basa en:

- El análisis y asociación de las acciones a ejecutar sobre las nuevas cimas creadas en la GSS. Estas acciones se obtienen de la tabla de análisis sintáctico en función del estado que se analiza y del token de entrada, y se asocian a los nodos en dichas cimas.
- La ejecución de las acciones asociadas a los nodos de las cimas de la GSS. Dichas acciones pueden crear, eventualmente, nuevas cimas, que serán instrumentadas como ya se ha indicado.

Durante el procesamiento de un token, el algoritmo ejecuta sucesivamente las acciones de reducción hasta que ya no es posible encontrar más acciones de dicho tipo activas, en la línea del método MVLR propuesto anteriormente. En este punto, es posible ejecutar todas las acciones de desplazamiento, y consumir efectivamente el token.

Tanto la GSS como el SPPF se construyen y relacionan eficientemente en función de las operaciones de desplazamiento y reducción de la siguiente forma:

- Acción de desplazamiento. Realizada sobre un nodo *O*, supone añadir a la GSS un segmento de la forma *O*<-*n*-*D*, donde: (i) *n* será una referencia a un nodo terminal para el símbolo en el SPPF, y (ii) *D* será un nodo correspondiente con el estado al que se transita en el desplazamiento.

- Acción de reducción. Realizada sobre un nodo *N* supone: (i) visitar tantos arcos en el camino que comienza en *N* como indique la reducción; la secuencia de nodos SPPF en sentido inverso τ representará la secuencia de nodos hijo en el SPPF implicada en la reducción, (ii) siendo *O* el nodo destino del último arco visitado, *A* la parte izquierda de la producción usada en la reducción, y *D* el nodo con el estado al que se transita, añadir a la GSS un segmento de la forma *O*<-*n*-*D*, con *n* una referencia al nodo en el SPPF para *A* con una lista de nodos hijo τ.

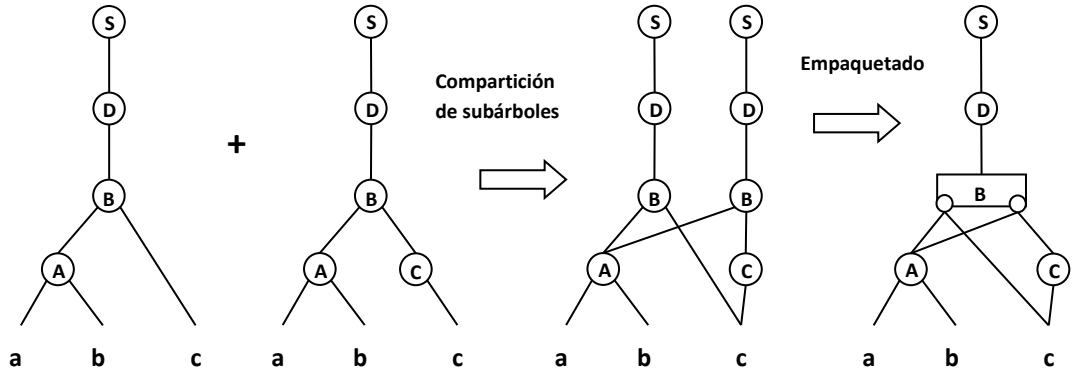


Figura 5.3.13. Compartición de subárboles y empaquetado por ambigüedad local.

A fin de optimizar el espacio utilizado por el bosque de análisis sintáctico, durante su construcción se produce una compartición de subárboles, lo que se traduce en que se trata de no crear nuevos nodos hijo que describen un árbol que ya existe, sino de referenciarlos. También trata de producirse un efecto de empaquetado por *ambigüedad local* en aquellos nodos que comparten la misma estructura de nodos padre, pero cada uno alberga un árbol diferente, eliminando redundancias de nodos a niveles superiores. En la Figura 5.3.13 se detalla el comportamiento de ambas optimizaciones.

```
PARSE(S0,ACCION,IR_A):
  N0 = nuevoNodoGSS(S0)
  U = {N0}
  mientras  U ≠∅
    ℜ  = U
    mientras ℜ≠∅
      REDUCE(ACCION,IR_A,U,ℜ)
    fin mientras
    DESPLAZA(ACCION,U,c)
    TokenSiguiente()
  fin mientras
  si hay un nodo N en la GSS y una acción accept en ACCION(estadoDe(N),EOF)
    devuelve OK
  si no
    devuelve ERROR
  fin si
```

Figura 5.3.14. Pseudocódigo para el algoritmo GLR (procedimiento PARSE).

El pseudocódigo de la Figura 5.3.14 detalla la estrategia seguida por la propuesta que Tomita presenta en [Tomita 1986] (véase [Tomita 1985] para la descripción original de Tomita). Como ya se ha sugerido, la estructura del mismo es similar a la del método MVLR. De esta forma, inicialmente se crea un nodo en la GSS asociado al estado inicial. Dicho nodo se utiliza para inicializar el conjunto *U* de *nodos activos*. El algoritmo itera, entonces, una fase de *reducción* seguida de una fase de *desplazamiento*, hasta llegar a un punto en el que no hay más nodos activos.

La fase de reducción realiza iterativamente todas las reducciones posibles sobre los nodos *a reducir,* almacenados en el conjunto $\Re$ (dicho conjunto se inicializa a los nodos activos), hasta que dicho conjunto se vacía. El proceso realizado en cada iteración de esta fase se detalla en la Figura 5.3.15. Dicho proceso actúa sobre el conjunto $\Re$, aplicando, sobre cada uno de los nodos, las posibles reducciones. Para ello, para cada uno de dichos nodos, obtiene las posibles acciones de reducción aplicables sobre el mismo (nótese que el algoritmo trabaja con tablas de análisis que pueden contener más de una acción en sus celdas), y aplica cada una de estas acciones. La aplicación de una acción reduce A ::= α sobre el nodo *N* supone seguir todos los posibles caminos de longitud $|\alpha|$ que parten en *N* (Figura 5.3.16), formando la secuencia τ de referencias a nodos SPPF en sentido inverso. Para cada nodo *N'* alcanzado:

```
REDUCE(ACCION,IR_A,U,ℜ):
  ℜ' = ∅
  para cada N en ℜ
    para cada reduce A ::= α en ACCION(estadoDe(N),Simbolo(tokenActual()))
      Θ = POPBODY(N,α)
      para cada <N', τ> en Θ
        si existe en la GSS un arco N'<-n-N'' tal que  EstadoDe(N'') =  IR_A(EstadoDe(N'),A)
          añadirHijos(n,τ)
        si no
          n = nuevoNodoSPPF(A, τ)
          si existe en U un nodo N'' tal que EstadoDe(N'') = IR_A(EstadoDe(N'),A)
            añadir a la GSS el arco N'<-n-N''
            ℜ' = ℜ' ∪ {N''}
          si no
            N'' = nuevoNodoGSS(IR_A(EstadoDe(N'),A))
            añadir a GSS el arco N'<-n-N''
            U  = U ∪ {N''}
            ℜ' = ℜ' ∪ {N''}
          fin si
        fin si
      fin para
    fin para
  fin para
  ℜ = ℜ'
```

Figura 5.3.15. Pseudocódigo para el algoritmo GLR (procedimiento REDUCE).

- Si ya existe un arco *N'*<-*n*-*N''*, siendo el estado asociado a *N''* aquel al que se transita por *A* en el autómata LR desde el asociado a *N'*, simplemente se añade la correspondiente secuencia de hijos τ al nodo SPPF al que apunta *n*.

- Si tal arco no existe, pero sí se ha generado tal nodo *N''* anteriormente en la fase de reducción, se crea dicho arco entre *N'* y *N''*, y se crea un nuevo nodo SPPF con *A* como símbolo y con la correspondiente secuencia τ como secuencia de nodos hijo. Así mismo, *N''* se considera de nuevo nodo a reducir, para tratar los nuevos caminos de reducción originados por el nuevo arco.

- En otro caso, se crea tal nodo *N''* y dicho arco en los términos indicados. Así mismo, ***N''*** se considera un nuevo nodo activo y también un nuevo nodo a reducir.

Como resultado de este proceso se obtendrá, por tanto, un nuevo conjunto de nodos a reducir, sobre el que se continuará iterando.

```
POPBODY(N,α):
  devuelve DOPBODY(N,|α|,[])
  donde DOPBODY(N,i,τ):
    si i=0  devuelve {<N,τ>}
    si no
      Resul = ∅
      para cada <N',n> tal que N' <-n-N está en la GSS
        Resul = Resul ∪ DOPBODY(N', i-1, n++τ)
      fin para
      devuelve Resul
    fin si
```

Figura 5.3.16. Pseudocódigo para la función POPBODY utilizada en el algoritmo GLR.

Por su parte, la fase de desplazamiento ejecuta todas las acciones de desplazamiento aplicables sobre los nodos activos, obteniendo, de esta forma, el conjunto de nodos activos para la siguiente iteración del algoritmo (Figura 5.3.17). Básicamente esta fase crea un nuevo nodo SPPF a partir del token actual, que será compartido por todos los arcos creados por los desplazamientos. En el conjunto de nuevos nodos activos habrá un nodo *N* asociado con cada estado diferente *s* que figure en las acciones *shift s* a ejecutar. De esta forma, todas aquellas acciones que involucren al nuevo estado crearán arcos al mismo nodo destino.

```
DESPLAZA(ACCION,U):
  U' = ∅
  n = nuevoNodoSPPF(tokenActual())
  para cada N en U
    si hay una acción shift s en ACCION(estadoDe(N), tokenActual())
      si existe en U' un nodo N' tal que EstadoDe(N') = s
        añadir a GSS el arco N<-n-N'
      si no
        N' = nuevoNodoGSS(s)
        añadir a GSS el arco N<-n-N'
        U' = U' ∪ {N'}
      fin si
    fin si
  fin para
  U = U'
```

Figura 5.3.17. Pseudocódigo para el algoritmo GLR (procedimiento DESPLAZA).

El algoritmo finaliza construyendo toda la GSS. La entrada será aceptada si dicha GSS contiene un nodo sobre el que es aplicable una acción de aceptación.

El algoritmo presentado se corresponde, de hecho, con la versión básica propuesta por Tomita. Dicha versión puede no funcionar correctamente sobre gramáticas con reglas *anulables por la derecha* (reglas del tipo A ::= αβ, donde β puede generar la cadena vacía). La fuente de incorrección se encuentra en la reutilización de estados que ya existen en la fase de reducción cuando se crean nuevos arcos debido a reducciones por producciones nulas. Una forma de resolver la misma es asociar con cada nodo activo un nivel, incrementar dicho nivel para nodos destino de arcos creados por reducciones nulas, y no permitir reutilizar nodos en niveles distintos a los de origen. No obstante, el algoritmo que resulta no termina sobre gramáticas con *recursión escondida a izquierdas* (gramáticas que albergan reglas de la forma A::=αβ, donde α puede generar la cadena vacía y β una forma sentencial que comienza por A). En [Nozohoor-Farshi 1991] se describe una reelaboración del algoritmo (componente reconocedor) que soluciona el problema con este tipo de gramáticas, y que es, por tanto, aplicable a gramáticas arbitrarias, aunque con un incremento en complejidad no desdeñable. Otra alternativa para resolver el problema del algoritmo de Tomita con las gramáticas con recursión escondida a izquierdas es la descrita en [Nederhof & Sarbo 1996]. Por último, el método RNGLR [Johnstone et al. 2004] aborda la corrección resolviendo el problema de la versión inicial del algoritmo con las gramáticas con reglas anulables por la derecha.

En cuanto a la construcción de bosques SPPF, la solución propuesta por Tomita para empaquetar nodos es llevar dicha acción cuando, tras una reducción, se encuentra que el correspondiente arco *O*<-*n*-*D* a añadir ya ha sido añadido anteriormente en la misma fase de reducción. En [Shaban 1994] se aborda en profundidad el aspecto de construcción de SPPFs en los métodos GLR. Otras alternativas a la construcción de bosques de análisis sintáctico, que producen representaciones más compactas, son las descritas en [Rekers 1992, Visser 1997]. No obstante, dichas representaciones difieren de las requeridas por las gramáticas de atributos multivista. Para finalizar, y a modo de ejemplo, la Figura 5.3.18 muestra la GSS y el SPPF construidos por dicho algoritmo para la gramática de la Figura 5.3.12 con entrada “e”.

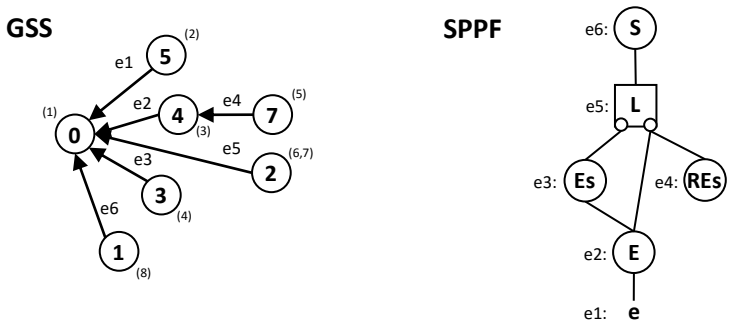


Figura 5.3.18. GSS y SPPF para la gramática y tabla de análisis de la Figura 5.3.12 con entrada “e”.

5.3.2.2 Aplicación del método GLR a la fase de análisis de las gramáticas multivista

En una primera aproximación puede plantearse el uso directo del algoritmo GLR para implementar eficientemente la fase de análisis en el modelo de ejecución de las gramáticas de atributos multivista. Para ello se considera la gramática global que resulta de unir las estructuras sintácticas de todas las vistas (tal y como se ha hecho en la Figura 5.3.12), y se aplica el algoritmo con las tablas de análisis obtenidas a partir de dicha gramática mediante alguno de los métodos de construcción de dichas tablas (p.e., el método LALR(1), tal y como muestra la misma Figura 5.3.12). Mientras que dicho enfoque funciona en algunos casos (p.e., para la estructura multivista de la Figura 5.3.8 y para la entrada "e", tal y como evidencia la Figura 5.3.18), para otros es, sin embargo, inapropiado, tanto desde el punto de vista de reconocimiento como desde el punto de vista de construcción de los bosques de análisis sintáctico requeridos por las gramáticas multivista.

Para ilustrar estos aspectos, se considerará la estructura sintáctica multivista de la Figura 5.3.20. La Figura 5.3.19 muestra el autómata LALR(1) para la gramática global obtenida a partir de dicha estructura multivista.

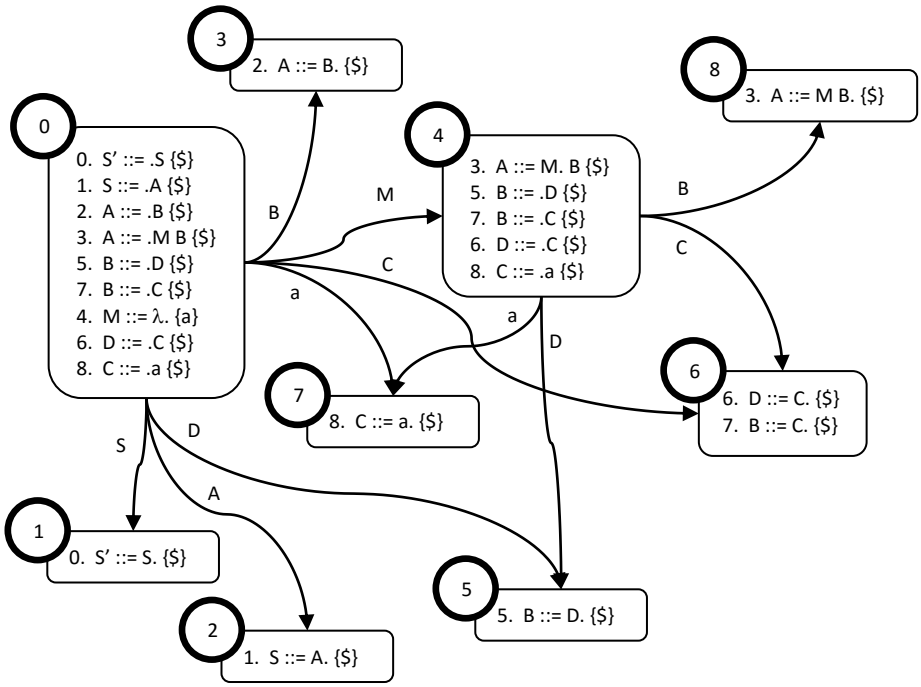


Figura 5.3.19. Autómata LALR(1) para la gramática (global) del ejemplo 2.

Vista 1	Vista 2
1. S ::= A	
2. A ::= B	3. A ::= M B 4. M ::= λ
5. B ::= D 6. D ::= C	7. B ::= C
8. C ::= a	

Figura 5.3.20. Estructura multivista de ejemplo 2.

La Figura 5.3.21 muestra la GSS y el SPPF generados por el análisis GLR (versión básica) durante el análisis de la única sentencia "a" generada.

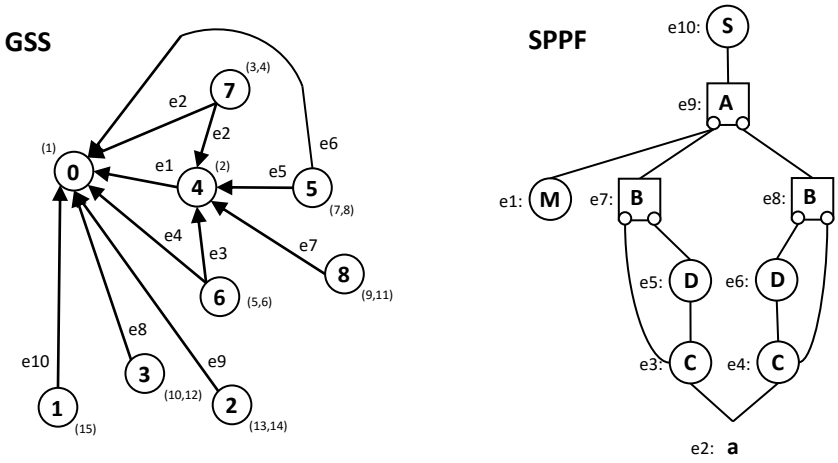


Figura 5.3.21. GSS y SPPF para la gramática de la Figura 5.3.20 con entrada "a".

La Figura 5.3.22 muestra, por último, el SPPF esperado (dicho SPPF es el resultado de *pegar* los dos árboles correspondientes a las dos estructuras sintácticas, como se pone también de manifiesto en dicha figura).

Como puede observarse en el ejemplo, los SPPFs obtenidos y esperados difieren substancialmente. Efectivamente:

- En el SPPF obtenido, los nodos B en cada vista *no* aparecen empaquetados. Efectivamente, dichos nodos podrían haberse colapsado en uno único, con varias listas de hijos.

- En este SPPF se observa también cómo existen duplicidades en la construcción de las estructuras. Efectivamente, las estructuras B-->C y B-->D-->C se han construido por duplicado, en lugar de compartirse una única estructura.

La combinación de ambos efectos redunda, de hecho, en que el SPPF resultante sea incluso incorrecto con respecto a la estructura multivista (en este caso, el B asociado a cada vista se estructura también como dicta la otra vista).

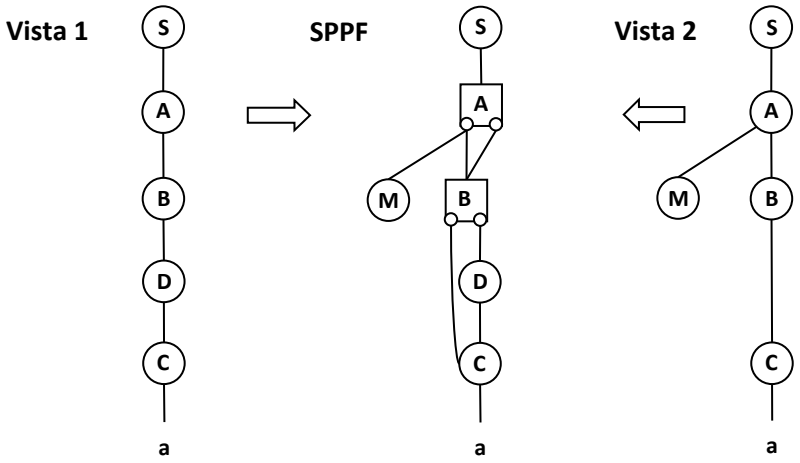


Figura 5.3.22. SPPF esperado para la gramática de la Figura 5.3.20 con entrada "a".

Para diluir estos efectos, pueden aplicarse heurísticas que, por una parte, mejoren el comportamiento de empaquetado, y, por otra, eliminen en la medida de lo posible las construcciones redundantes. Obsérvese que, de hecho, el aspecto más crítico es mejorar el empaquetado, ya que las estructuras redundantes siempre pueden filtrarse al añadirse las listas de nodos hijo (basta con comprobar, al añadir una secuencia de nodos hijo, que dicha secuencia no se haya añadido ya literalmente, o que no exista una lista de nodos hijo cuyos símbolos coincidan con los de aquella, ya que esto evidenciaría la aplicación redundante de la misma producción).

En esta tesis, para mejorar el empaquetado hemos ensayado las siguientes heurísticas:

- Al aplicar una acción de reducción, todos los arcos generados comparten la referencia al mismo nodo en la SPPF.
- Es posible llevar la cuenta de la *profundidad* de los nodos en cada fase de reducción en la SPPF (los nodos empaquetados podrán tener varias de estas *profundidades*). De esta forma, si se desea crear un nodo para *A* con profundidad *p* que ya existe, puede utilizarse una referencia a dicho nodo en su lugar.

Estas heurísticas mejoran substancialmente los esfuerzos previos citados en [Temprado et al. 2010-a] y descritos en [Temprado 2010], y se comportan satisfactoriamente en la práctica totalidad de los casos realistas. Por ejemplo, la primera de las heurísticas será suficiente siempre y cuando consiga aislar cada reducción para cada regla en un único estado, cosa que ocurre en la mayoría de los casos (en concreto, esta heurística permite tratar adecuadamente el ejemplo de la Figura 5.3.20 obteniendo el bosque mostrado en la Figura 5.3.22). En casos más atípicos es posible reforzarla con la heurística basada en la profundidad, o bien utilizar únicamente esta segunda heurística. No obstante, tratándose de heurísticas siempre es posible encontrar situaciones en las que no se comporten adecuadamente.

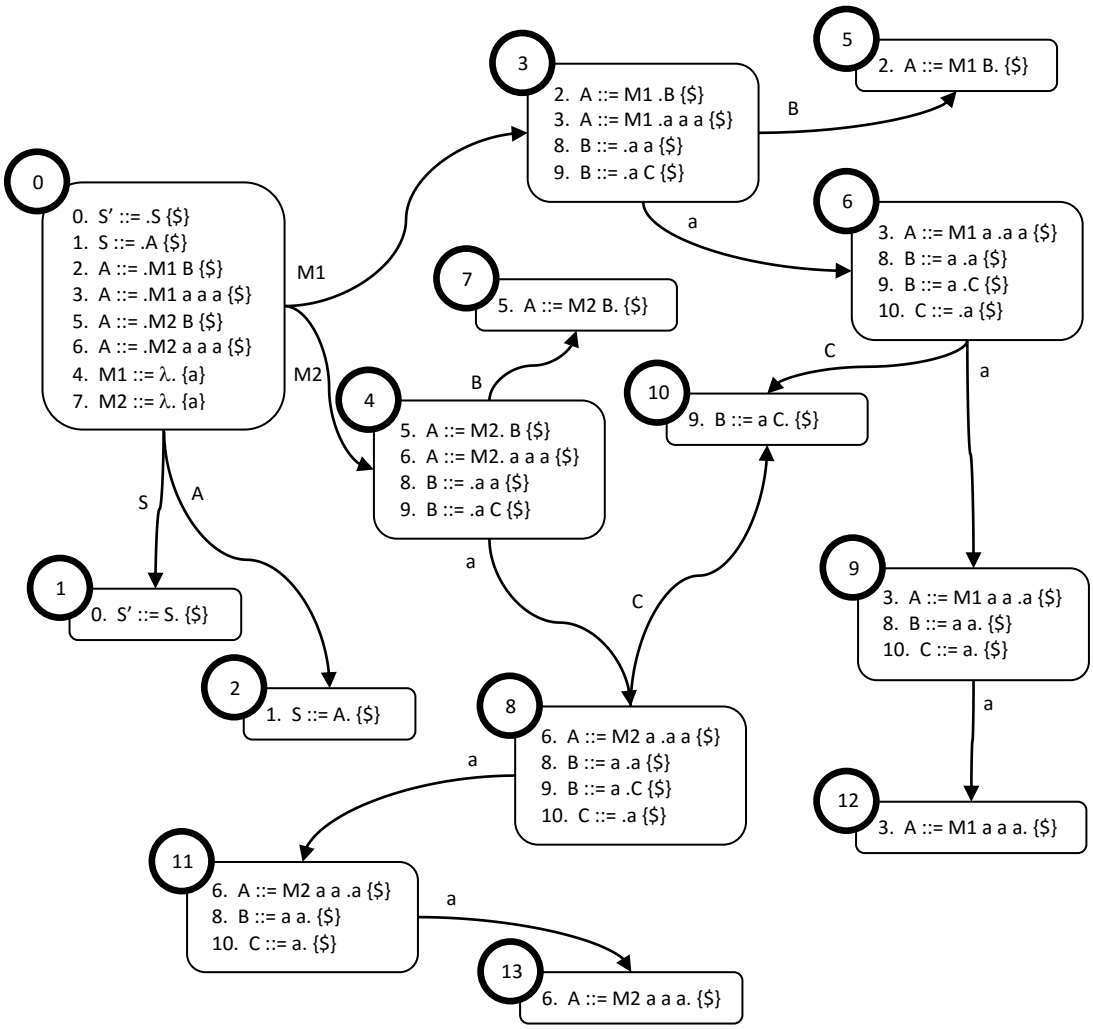


Figura 5.3.23. Autómata LALR(1) para el caso problemático.

Por ejemplo, considérese la estructura multivista que se muestra en la Figura 5.3.24, así como el correspondiente autómata LALR(1) para la gramática global en la Figura 5.3.23.

Vista 1	Vista 2
1. S ::= A	
2. A ::= M1 B 3. A ::= M1 a a a 4. M1 ::= λ	5. A ::= M2 B 6. A ::= M2 a a a 7. M2 ::= λ
8. B ::= a a	9. B ::= a C 10. C ::= a

Figura 5.3.24. Estructura multivista de caso problemático.

Para el análisis de "aa":

- El éxito o el fracaso de la primera de las heurísticas dependerá del orden de las reducciones. Así, por ejemplo, si primero se aplican las dos reducciones para B ::= a a, se generará incorrectamente el bosque que se muestra en la Figura 5.3.25.a, frente al esperado, mostrado en la Figura 5.3.25.b. Si, por ejemplo, comienza reduciéndose B ::= a C, el empaquetado se producirá correctamente.

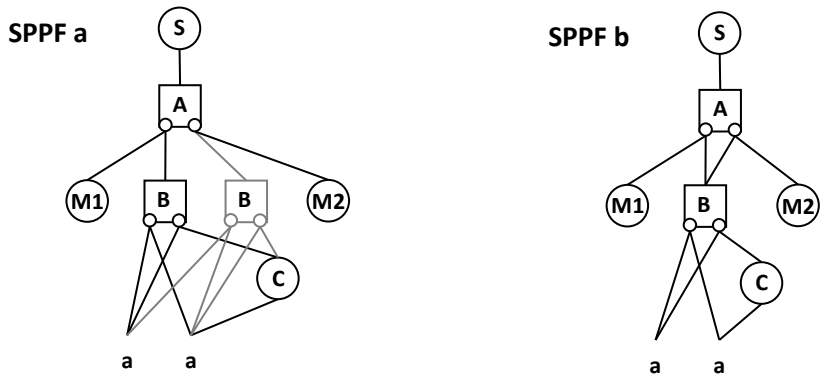


Figura 5.3.25. Bosque para el caso problemático por la primera heurística (inadecuado) (a) y el esperado (b).

- La aplicación de la segunda heurística, aisladamente, tampoco resuelve correctamente el caso, al tener asignado los nodos para B diferentes profundidades (Figura 5.3.26).

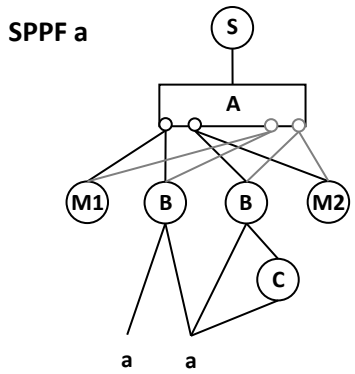
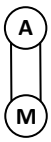
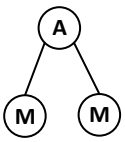


Figura 5.3.26. Bosque para el caso problemático por la segunda heurística (inadecuado).

- Por último, la aplicación combinada de ambas heurísticas sí logra resolver adecuadamente el caso. Sin embargo, debe tenerse en cuenta que la segunda heurística puede conducir, bien a empaquetar incorrectamente nodos diferentes para reducciones por producciones nulas (p.e., considérese la gramática: A ::= M M, M ::= λ de la Figura 5.3.27), en caso de que se decida aplicar la heurística también a las reducciones por producciones nulas, bien a no compartir adecuadamente nodos creados por producciones vacías (p.e., considérese la estructura multivista de la Figura 5.3.27), en caso de que decida no aplicarse la heurística a dichas reducciones.

Gramática de reglas anulables	SPPF incorrecto	SPPF correcto
A ::= M M M ::= λ		

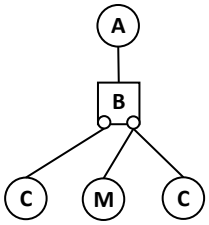
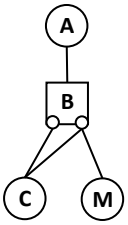
Vista 1	Vista 2	SPPF incorrecto	SPPF correcto
	S ::= B		
B ::= C	B ::= M C M ::= λ		
	C ::= λ		

Figura 5.3.27. Casos problemáticos por reglas anulables.

5.3.2.3 El algoritmo MVGLR

El principal atractivo de la aplicación directa del método GLR a la fase de análisis es la gestión de la pila de análisis, que se bifurca cuando se encuentran caminos de análisis divergentes, y se reunifica cuando dichos caminos convergen. Este tipo de comportamiento es, precisamente, el que se busca para evitar las potenciales redundancias en los *n* procesos de análisis simultáneos mantenidos por el método MVLR. Sin embargo, como contrapartida el algoritmo GLR debe pervertirse para emular dicho comportamiento mediante la gestión del no determinismo llevado a cabo por el mismo. Para evitar las consecuencias de esta perversión (el ajuste del método no deja de tener un carácter heurístico, y, por tanto, incompleto, al contrario que el algoritmo MVLR, que funciona adecuadamente para cualquier gramática multivista), hemos llevado a cabo una reformulación del método MVLR para trabajar sobre una GSS en lugar de sobre *n* pilas simultáneas. Dicha reelaboración requiere, primeramente, tablas de análisis unificadas adecuadas, que hagan explícitos los puntos en los que coinciden y difieren las distintas gramáticas. A continuación, desarrollamos los detalles.

5.3.2.3.1 Multiautómatas LR

Las tablas de análisis asociadas a la gramática global que resultan de unir las distintas subgramáticas, además de caracterizar el proceso de análisis simultáneo para cada una de las vistas, caracterizan también el análisis respecto a los *entrelazados* de dichas vistas. Como resultado, el número de caminos posibles de análisis aumenta significativamente (incluso de forma exponencial). Aunque la estrategia de tabulación del método GLR pueda manejar adecuadamente dicha explosión combinatoria, la complejidad del análisis puede aumentar en un par de órdenes (la complejidad asintótica de los algoritmos de análisis generales es cúbica, frente a la complejidad lineal de métodos como el LR y, como consecuencia, como el MVLR).

Una forma de evitar el fenómeno de entrelazado es basar la construcción de las tablas, no en el autómata asociado a la gramática global, sino en los autómatas individuales asociados a cada una de las vistas. Para ello, dichos autómatas pueden representarse conjuntamente mediante un único grafo de transiciones. En dicho grafo se compartirán aquellas partes que resulten comunes entre los autómatas individuales. La estructura resultante se denominará *multiautómata* LR, y representará, no un único autómata LR, sino *n* autómatas que comparten aquellas partes que tienen en común. Dicha estructura puede utilizarse para generar unas tablas unificadas en la forma habitual. De hecho, tales tablas permitirán el análisis LR respecto a cada gramática individual, pero, al hacer explícita la relación entre las distintas gramáticas, también serán de utilidad para orquestar el análisis simultáneo con respecto a cada una de las vistas. De hecho, las tablas unificadas pueden substituir a las *n* tablas IR_As y a las *n* tablas ACCIONes utilizadas en la formulación del MVLR (no detallaremos, aquí, por ser directas, las modificaciones necesarias). En la construcción de dichas tablas resulta también útil determinar, para cada estado, a qué gramática o gramáticas está asociado dicho estado. Esto se llevará a cabo construyendo una tabla Φ que, dado un estado, determinará el conjunto de gramáticas al que éste está asociado.

En la construcción del multiautómata LR es necesario caracterizar de manera precisa qué estados pueden compartirse. Por supuesto que los estados a compartir deberán ser *fusionables* (en un sentido que dependerá del método utilizado en la construcción de los autómatas, método que se asume el mismo para todas las vistas). Además, será necesario asegurar que, para cada transición, se transite en los autómatas implicados a estados que puedan compartirse. Más precisamente:

- El multiautómata puede construirse incrementalmente, añadiendo autómatas al mismo.
- Dado un multiautómata LR *M* y un nuevo autómata LR *A* a añadir a *M*, es posible caracterizar los *estados compartibles* entre *M* y *A* como aquellos que son fusionables y desde los que, además, siempre se transita a estados compartibles entre sí. Más precisamente, el conjunto de pares de estados compartibles **COMP**(*M*,*A*) vendrá dado por la mínima solución de:

$$\textbf{COMP}(M,A) = \{(s,s') \mid s \in \textbf{estados de } M,\ s' \textbf{ estado de } A,\ \text{Fusionables}(s,s'),\ s\text{-X->}s'',\ s'\text{-X->}s''' \Rightarrow (s'', s''') \in \textbf{COMP}(M, A)\}$$

La Figura 5.3.28 esboza el algoritmo de construcción del multiautómata LR *M* a partir de los *n* autómatas A_1, …, A_n asociados con las distintas vistas. Básicamente, dicho algoritmo va añadiendo sucesivamente los autómatas individuales a dicho multiautómata. El añadido de un autómata A_i a *M* supone computar, primeramente, **COMP**(*M*,A_i). Este conjunto puede determinarse utilizando una estrategia de punto fijo estándar. Efectivamente:

- El algoritmo mantiene un conjunto de pares de estados *candidatos* a ser compartidos. Inicialmente el conjunto de pares candidatos viene dado por los pares de estados fusionables.
- El algoritmo, entonces, refina iterativamente dicho conjunto hasta que no se producen cambios. En cada iteración considera los pares de estados candidatos a ser compartidos, y aplica la condición básica de compartibilidad. Si dicha condición se viola, el par se elimina del conjunto de candidatos.

Cuando el proceso termina, en el conjunto de candidatos aparecerán todos los pares de estados que pueden compartirse en el multiautómata (es decir, el conjunto buscado **COMP**(*M*,A_i)). Entonces, es posible actualizar el multiautómata, añadiendo al mismo todos aquellos estados de A_i que no han sido compartidos, todas las transiciones entre estados no compartidos, y las transiciones con origen en estados no compartidos y destino en estados compartidos (las transiciones con origen en estados compartidos deben tener como destino estados compartidos, y, por tanto, estarán ya presentes en la configuración anterior del multiautómata que se está construyendo). Además, si procede, debe fusionarse la información de los estados compartidos en A_i con la de los de *M*.

Entrada: *N* Autómatas LR $A_1 \ldots A_n$
Salida: Un multiautómata *M* que representa conjuntamente a $A_1 \ldots A_n$
Método:
estados de *M* = ∅
transiciones de *M* = ∅
para i=1 **hasta** *n*
Candidatos = {(s,s') | s ∈ **estado de** *M*, s'∈ **estado de** A_i y Fusionables(s,s')}
repetir
CandidatosEliminados = **false**
para cada (s,s') ∈ Candidatos
SonCompartibles = **true**
para cada transición X **con origen en** s, **y mientras** SonCompartibles
sea (s'',s''') **tal que** s-X->s'' y s'-X->s'''
si (s'',s''') ∉ Candidatos
SonCompartibles = **false**
Candidatos = Candidatos – {(s,s')}
fin si
fin sea
fin para
CandidatosEliminados = CandidatosEliminados **o no** SonCompartibles
fin para
mientras CandidatosEliminados
Compartidos = {s|∃s':(s',s)∈Candidatos}
estados de M = **estados de** M ∪ {*s* | *s* ∈ **estados de** A_i y s ∉ Compartidos}
fusionar Compartidos **con estados de** M
transiciones de M = **transiciones de** M ∪ {s -X->s'|s-X->s' ∈ **transiciones de** A_i y
s ∉ Compartidos y s' ∉ Compartidos}
∪ {*s*-X->s'' | s-X->s' ∈ **transiciones de** A_i y
s ∉ Compartidos y (s'',s') ∈ Candidatos}
fin para

Figura 5.3.28. Pseudocódigo para la construcción del multiautómata LR asociado a *n* autómatas LR $A_1 \ldots A_n$.

La condición de fusionabilidad entre estados dependerá del método utilizado para construir los autómatas. A este respecto:

- Si el método de construcción utilizado para los autómatas individuales es el método LR(0) o el método LR(1), los estados compartidos en el multiautómata obtenido se corresponderán con los correspondientes estados en los autómatas de las vistas, ya que dichos estados compartidos serán idénticos en los distintos autómatas.

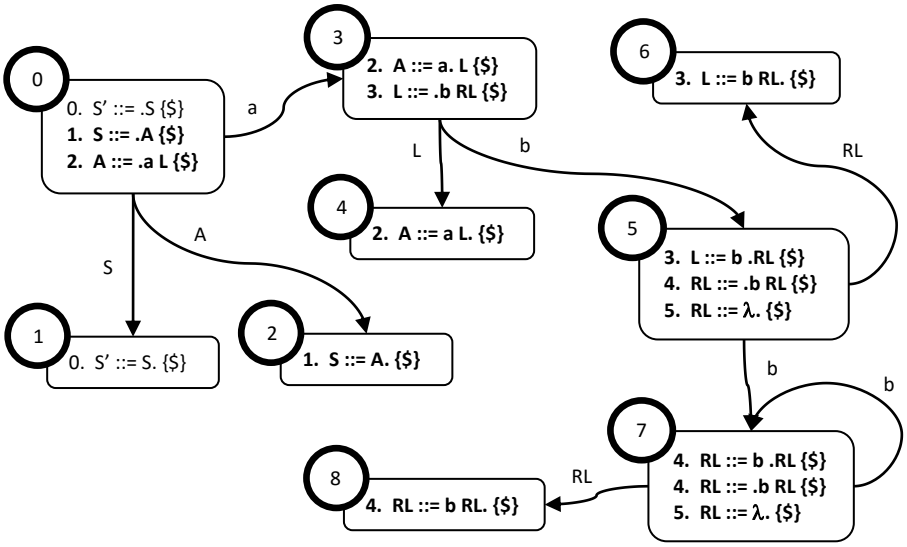
- Por el contrario, si el método utilizado es el LALR(1), los estados compartidos se corresponderán con la *fusión* de los estados individuales (en el sentido de fusionar estados con corazones comunes, uniendo los símbolos de preanálisis de los elementos). A este respecto, podemos conjeturar que, en virtud de la conformidad entre las distintas gramáticas, también en este caso los estados compartidos se corresponderán con los de los autómatas individuales (en el sentido de que los estados con corazones comunes en los distintos autómatas serán, de hecho, idénticos). Esto es debido a que, en último término, los conjuntos de preanálisis en los estados con corazones comunes se deberán, bien a la estructura interna de las subgramáticas, no dependiendo de la forma en la que los estados se generan, bien a los símbolos de núcleo y a los terminales que pueden seguir a estos en las posibles derivaciones más a la derecha. Dado que, en este último caso, las estructuras de las vistas coinciden en lo que se refiere a las secuencias de símbolos de núcleo y terminales, dichos contextos se mantendrán de vista a vista. No obstante, este hecho es una mera conjetura cuya prueba queda fuera del alcance de esta tesis, ya que, por otra parte, no afecta al comportamiento de análisis final, al resultar, en cualquier caso, los conjuntos de preanálisis superconjuntos de los fusionados. De esta forma, y en caso de resultar falsa la conjetura realizada, habría que aplicar un filtro adicional a las gramáticas de las vistas, en el sentido de no permitir la introducción de conflictos en el multiautómata final.

<table>
<tr><th>Vista 1</th><th>Vista 2</th></tr>
<tr><td colspan="2">1. S ::= A</td></tr>
<tr><td>2. A ::= a L</td><td>3. A ::= B L
4. B ::= a</td></tr>
<tr><td colspan="2">5. L ::= b RL
6. RL ::= b RL
7. RL ::= λ</td></tr>
</table>

Figura 5.3.29. Estructura multivista de ejemplo.

A modo de ejemplo, considérese la estructura multivista de la Figura 5.3.29. La Figura 5.3.30 muestra los autómatas LALR(1) individuales, y la Figura 5.3.31 muestra el correspondiente multiautómata. Obsérvese que, en este caso la conjetura realizada respecto a los autómatas LALR(1) de las distintas vistas se cumple, al resultar los estados con el mismo corazón idénticos en ambas vistas.

Autómata LALR(1) para la vista 1



Autómata LALR(1) para la vista 2

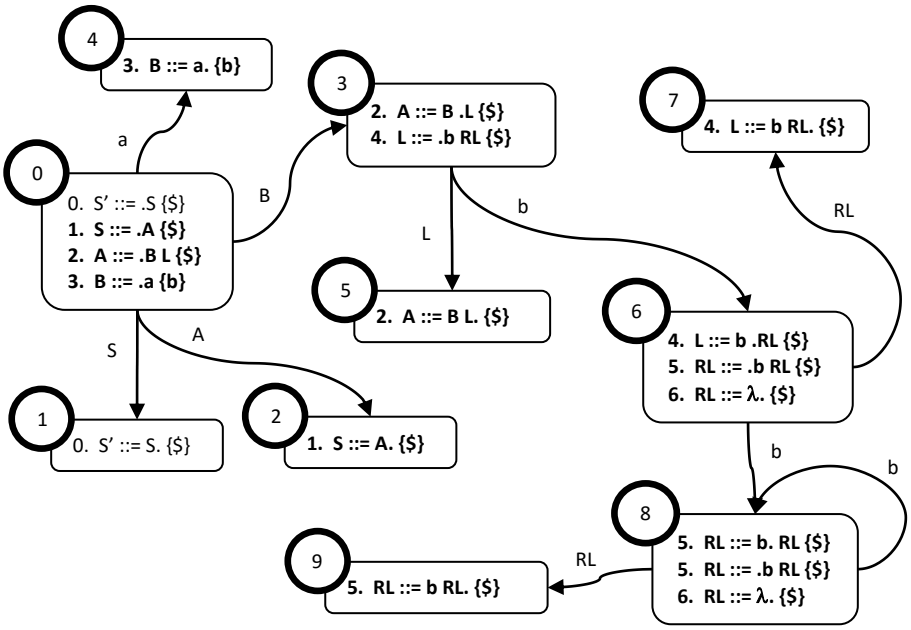


Figura 5.3.30. Autómata LALR(1) para la vista 1 y vista 2.

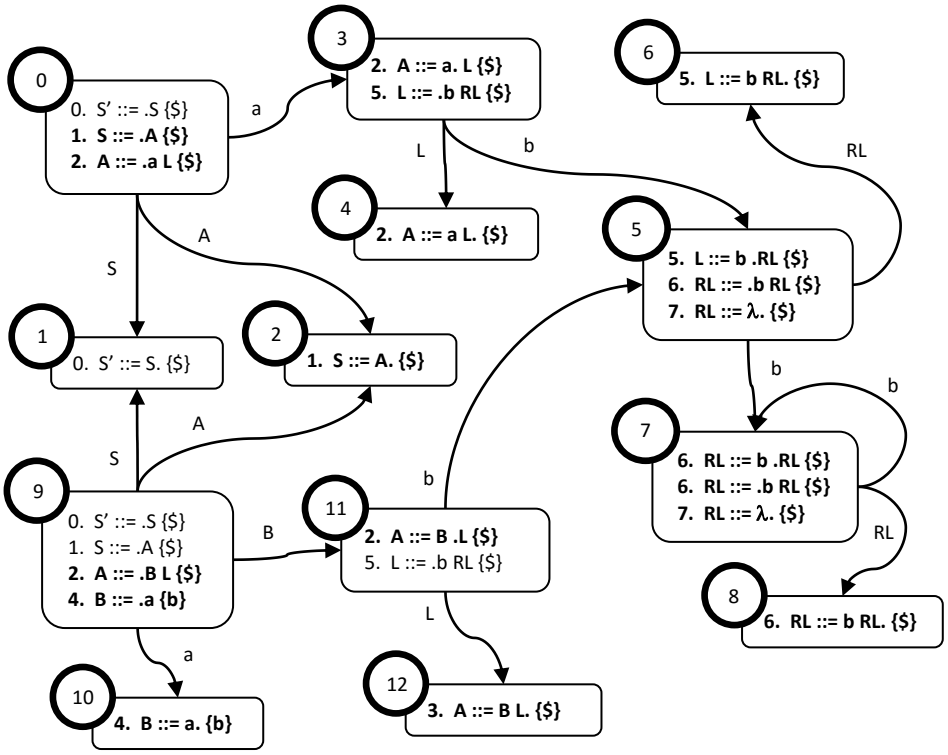


Figura 5.3.31. Multiautómata a partir de los autómatas individuales de vista 1 y vista 2.

La Figura 5.3.32 muestra, por último, las tablas de análisis obtenidas a partir de dicho multiautómata, así como la tabla de clasificación de estados Φ. Como se señala también en dicha figura, las tablas de análisis fusionan en unas únicas las tablas de análisis que se obtienen de cada autómata individual, haciendo explícita la compartición de las partes comunes. De esa forma, fijando un estado inicial apropiado, dichas tablas pueden ser utilizadas sin problemas para el análisis LR con respecto a cada una de las vistas, tal y como se ha indicado anteriormente, y, por tanto, como tablas unificadas para guiar al algoritmo MVLR. Reseñar, por último, que, al contrario de las obtenidas a partir de la gramática global, estas tablas estarán siempre libres de conflictos (en caso contrario, la especificación multivista sería rechazada como no válida).

Reglas Gramaticales:

1. **S ::= A**
2. **A ::= a L**
3. **A ::= B L**
4. **B ::= a**
5. **L ::= b RL**
6. **RL ::= b RL**
7. **RL ::= λ**

Φ

Estado	Vistas
0	1
1	1, 2
2	1, 2
3	1
4	1
5	1, 2
6	1, 2
7	1, 2
8	1, 2
9	2
10	2
11	2
12	2

Estado	a	b	$	S	A	B	L	RL
0 (inicial)	Shift 3			1	2			
1			Accept					
2			Reduce 1					
3		Shift 5					4	
4			Reduce 2					
5		Shift 7	Reduce 7					6
6			Reduce 5					
7		Shift 7	Reduce 7					8
8			Reduce 6					
9 (inicial)	Shift 10			1	2	11		
10		Reduce 4						
11		Shift 5					12	
12			Reduce 3					

Figura 5.3.32. Tabla de análisis sintáctico LALR(1) del multiautómata de la Figura 5.3.31.

5.3.2.3.2 El algoritmo

Como ya se ha indicado, el algoritmo MVGLR es una reformulación directa del algoritmo MVLR que trabaja sobre una GSS. Dicho algoritmo está guiado por las tablas obtenidas a partir del correspondiente multiautómata LR. De esta forma, las principales diferencias entre el algoritmo MVGLR y el MVLR son:

- En lugar de manejar un conjunto fijo de *n* pilas, se maneja un conjunto variable de nodos GSS activos, que, como en el caso del algoritmo GLR, aumenta o disminuye conforme la GSS se bifurca o se reunifica. En el método MVGLR, la bifurcación de la GSS está asociada a las reducciones. Por su parte, las reunificaciones de las pilas están asociadas a los desplazamientos, y a las reducciones de núcleo. En particular, el algoritmo evita la problemática reutilización de estados en la fase de reducción, ya que, debido a las características de las diferentes gramáticas (que, bien comparten totalmente subgramáticas asociadas a los símbolos de núcleo, bien difieren totalmente en dichas subgramáticas), dicha reutilización sería anecdótica.

- Como consecuencia, los procesos de análisis no son independientes, sino que, en virtud de la estructura GSS, pueden compartir los tramos comunes.

```
PARSE(S0s,ACCION,IR_A,Φ, Θ, N):
  Π = ∅
  para cada S0 en S0s
    Π = Π ∪ {nuevoNodoGSS(S0)}
  fin para
  repetir
    repetir
      REDUCENONUCLEO(Π,ACCION,IR_A, Φ, Θ, N)
      reduccionesRealizadas = REDUCENUCLEO(Π,ACCION,IR_A, Φ,Θ)
    hasta no reduccionesRealizadas
    si (∃π∈Π: ACCION(EstadoDe(π),Simbolo(TokenActual())) = accept)
      devuelve OK
    fin si
    si (∃π∈Π: ACCION(EstadoDe(π),Simbolo(TokenActual())) = ⊥)
      devuelve ERROR
    fin si
    DESPLAZA(ACCION,Π)
    SiguienteToken()
  fin repetir
```

Figura 5.3.33. Pseudocódigo para el algoritmo MVGLR (procedimiento PARSE).

El pseudocódigo de la Figura 5.3.33 muestra la estrategia de alto nivel del método. Como en los otros casos, el método se modeliza como un procedimiento *PARSE*, que, al igual que para el método MVLR, toma como parámetros (i) el conjunto de estados iniciales (S0s), (ii) el número de gramáticas (NGs), (iii) la función informante de construcciones (Θ), y (iv) la función informante de núcleos (N). Así mismo, al contrario que dicho método, toma como parámetro una única tabla ACCION, y una única tabla IR_A (aunque, como ya se ha indicado anteriormente, es directo modificar el método MVLR para que utilice las tablas asociadas al multiautómata, y, por tanto, para que también coincida con el MVGLR en este punto) Por último, también toma como parámetro la tabla de asignación de estados a gramáticas Φ.

De esta forma, con la única diferencia de que, en lugar de un vector fijo de pilas, Π representa un conjunto de nodos GSS (las cimas de las pilas vigentes en cada momento del análisis), que, en la Figura 5.3.33 se inicializa con los nodos correspondientes a los estados iniciales del multiautómata, la estructura es análoga a la del algoritmo MVLR. Este hecho también es cierto con respecto a la lógica que lleva a cabo las reducciones *no* de núcleo (Figura 5.3.34): dicha lógica es análoga a la integrada en el algoritmo MVLR, con la diferencia de manejar conjuntos de nodos GSS que representan cimas de pilas vigentes, en lugar del vector fijo de pilas ya aludido anteriormente. En particular, Π puede cambiar como consecuencia de la realización de las reducciones. Por tanto, el proceso descrito en la Figura 5.3.34 es un proceso iterativo que itera hasta que no existen más nodos en Π sobre los que pueden realizarse reducciones no de núcleo (en el caso del MVLR, se iteraba hasta que no podía aplicarse ninguna de estas reducciones sobre la cima de ninguna de las *n* pilas). En dicha

iteración se determina, además, el nuevo contenido de Π una vez agotadas las reducciones no de núcleo (Π', en el pseudocódigo). De esta forma, en cada iteración:

```
REDUCENONUCLEO(Π,ACCION,IR_A,Φ, Θ,N):
  Π' = ∅
  mientras Π≠∅
    extraer un elemento π de Π
    si ACCION(EstadoDe(π),Simbolo(TokenActual())) = reduce A ::= α y no N(A)
      Γ = POPBODY(Π[i],α)
      para cada <N',τ> ∈ Γ
        si  (∃g ∈ Φ(EstadoDe(N')): Θ(g,A::=α))
          n = nuevoNodoSPPF(A,τ)
        si no
          n = ⊥
        fin si
        N'' = nuevoNodoGSS(IR_A(EstadoDe(N'),A))
        añadir a GSS el arco N'<-n-N''
        Π = Π ∪ {N''}
      fin para
    si no
      Π' = Π' ∪ {π}
    fin si
  fin mientras
  Π = Π'
```

Figura 5.3.34. Pseudocódigo para el algoritmo MVGLR (procedimiento REDUCENONUCLEO).

- Se extrae un nodo π de Π, y se determina si la acción aplicable sobre dicho nodo es una reducción *no* de núcleo (nótese que, dado que las tablas utilizadas se han generado a partir del correspondiente multiautómata LR, existirá, a lo sumo, una única acción en la correspondiente celda).

- Se aplica la reducción. En este caso, como en el caso del GLR, el resultado no es único, sino que puede afectar a múltiples caminos, en virtud de las bifurcaciones y reunificaciones en la GSS. El método de reducción es, por tanto, análogo al usado en el GLR, con la salvedad de que no se reutilizan estados ya generados en la fase de reducción. Así mismo, para cada reducción realizada, se utiliza la tabla de asignación de estados a gramáticas Φ para determinar la gramática o gramáticas a la(s) que corresponde el estado descubierto, origen de la transición, así como la función Θ para determinar si la producción utilizada en la reducción es constructora en alguna de dichas gramáticas. En caso afirmativo, se procede a construir el correspondiente nodo en el SPPF.

La aplicación de las reducciones de núcleo sigue también una estructura análoga a la de la correspondiente etapa en el algoritmo MVLR (Figura 5.3.35). Efectivamente, en caso de que sea aplicable una reducción de dicho tipo sobre alguno de los nodos cima, en el resto de los nodos también será aplicable la correspondiente reducción. Por tanto, se procede a ejecutar dichas reducciones sobre cada nodo cima. Así mismo, se construye el nodo compartido por los arcos creados por todas las reducciones realizadas, añadiendo las listas de hijos para aquellas reducciones constructoras. Por último, esta etapa tiene en cuenta también la compartición del mismo nodo destino para todas aquellas reducciones que suponen transitar al mismo estado, y, por tanto, la potencial reunificación de varias pilas en una única.

```
REDUCENUCLEO(Π,ACCION,IR_A,Φ, Θ):
  si (∃π∈Π: ACCION(EstadoDe(π),Simbolo(TokenActual())) = reduce A ::= α)
    n = nuevoNodoNoTerminal(A)
    Π' = ∅
    para cada π∈Π
      reduce A ::= α =  ACCION(EstadoDe(π),Simbolo(TokenActual()))
      Γ = POPBODY(π,α)
      para cada <N',τ> ∈ Γ
        si (∃g ∈ Φ(EstadoDe(N')):  Θ(g,A::=α))
          añadeHijos(n, τ)
        fin si
        si existe en Π' un nodo N'' tal que EstadoDe(N'') = IR_A(EstadoDe(N'),A)
          añadir a la GSS el arco N'<-n-N''
        si no
          N'' = nuevoNodoGSS(IR_A(EstadoDe(N'),A))
          añadir a la GSS el arco N'<-n-N''
          Π' = Π' ∪ {N''}
        fin si
      fin para
    fin para
    Π = Π'
    devuelve cierto
  fin si
  devuelve falso
```

Figura 5.3.35. Pseudocódigo para el algoritmo MVGLR (procedimiento REDUCENUCLEO).

Por último, la etapa de desplazamiento es análoga a la descrita para el método GLR (Figura 5.3.17).

La Figura 5.3.36 muestra el funcionamiento del algoritmo MVGLR para el ejemplo de la Figura 5.3.29 y para la entrada “abb”.

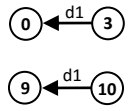
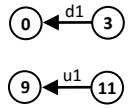
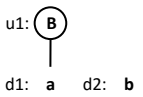
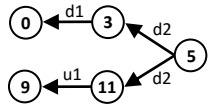
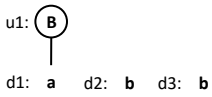
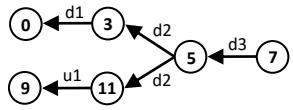
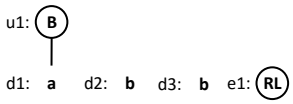
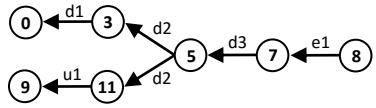
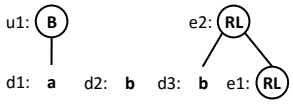
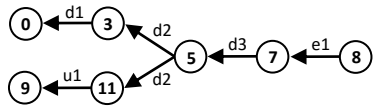
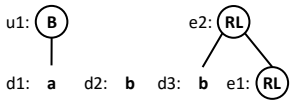
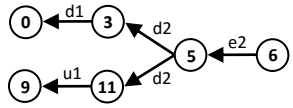
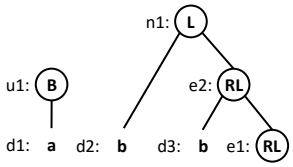
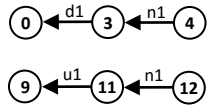
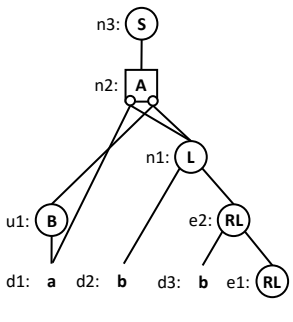
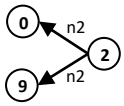
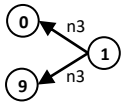
Entrada	GSS (parte activa)	Acción	SPPF
a		(0, d1): Shift 3 (9, d1): Shift 10	
b		(10, u1): Reduce 4 "**B ::= a**" (9, B): 11	
		(3, d2): Shift 5 (11, d2): Shift 5	
b		(5, d3): Shift 7	
$		(7, e1): Reduce 7 "**RL ::= λ**" (7, RL): 8	
		(8, e2): Reduce 6 "**RL ::= b RL**" (5, RL): 6	
		(8, e2): Reduce 6 "**RL ::= b RL**" (5, RL): 6	
		(6, n1): Reduce 5 "**L ::= b RL**" (3, **L**): 4 (11, **L**): 12	
		(4, n2): Reduce 2 " **A ::= a L**" (0, **A**): 2 (12, n2): Reduce 3 " **A ::= B L**" (9, **A**): 2	
		(2, n3): Reduce 1 "**S ::= A**" (0, **S**): 1 (9, **S**): 1	
		(1): Accept	

Figura 5.3.36. Ejemplo de proceso de análisis y construcción de bosques de MVGLR (parte 1).

Obsérvese, por último, que los dos métodos presentados, MVLR y MVGLR, son equivalentes entre sí, difiriendo únicamente en el uso de un número fijo de pilas o en el uso de una estructura más adaptativa y flexible. Mientras que la complejidad asintótica de ambos métodos coincide (lineal en relación con el tamaño del documento), ambos métodos son, en cierta forma, complementarios, en el sentido de que las ventajas de uno se traducen en los inconvenientes del otro. Efectivamente:

- Por una parte, el método MVLR puede presentar tramos redundantes de análisis, debido a que los procesos de análisis simultáneo son independientes entre sí, con la salvedad de los puntos de sincronización en los desplazamientos y en las reducciones de núcleo. Esta redundancia es paliada por el método MVGLR, que permite la reunificación de distintos caminos de análisis en los citados puntos de sincronización, aplicando técnicas análogas a las utilizadas en los métodos GLR.

- Por otra parte, la gestión de las estructuras de datos en el método MVLR es más simple y ágil que en el método MVGLR. A nivel de implementación, mientras que la gestión de las *n* pilas simultáneas es simple, ya que el espacio para dichas pilas puede estar asignado a priori, la gestión de la GSS radica, en último término, en mecanismos de gestión de memoria dinámica, lo que puede incidir negativamente en el rendimiento final del algoritmo.

De esta forma, en escenarios que requieran procesar documentos muy grandes (p.e., escenarios *bigdata*), o en los que existan solapes significativos entre vistas, será más apropiado el método MVGLR, mientras que en escenarios que requieran procesar documentos pequeños (p.e., generación de aplicaciones a partir de especificaciones descritas en forma de documentos XML), o involucren vistas mayoritariamente independientes entre sí, el método MVLR puede ser el método de elección.

5.3.3 Aspectos de implementación

Para implementar de manera efectiva la fase de análisis de gramáticas multivista para el procesamiento de documentos XML mediante alguno de los métodos propuestos, es necesario abordar los dos siguientes aspectos:

- Integración con un marco de procesamiento XML. Siguiendo las directrices expuestas en el Capítulo 3, para evitar las complejidades del procesamiento de los documentos XML a bajo nivel, es conveniente integrar la fase de análisis con un marco de procesamiento XML de propósito general. Dicha integración puede llevarse a cabo proporcionando un escáner XML, que produzca una vista de los documentos XML como secuencias de tokens. De esta forma, la integración es especialmente conveniente con marcos orientados a eventos, como SAX o StAX. La sección 5.3.3.1 aborda, a modo de ejemplo, la integración con SAX (la integración con StAX es análoga a la ya descrita en el Capítulo 3).

- Construcción de los bosques de análisis sintáctico atribuidos. Será necesario proporcionar el soporte necesario para construir los bosques de análisis sintácticos atribuidos que sean entrada para la fase de evaluación. Las secciones 5.3.3.2 y 5.3.3.3 ejemplifican una posible solución.

Una vez resueltos los aspectos anteriores, la fase de análisis estará gobernada por el algoritmo de análisis elegido, que, a su vez, operará sobre el marco XML subyacente para obtener los elementos de información necesarios (vistos como tokens), y sobre el componente de construcción de bosques atribuidos para generar las estructuras necesarias para la fase de evaluación.

5.3.3.1 Ejemplo de integración con un marco XML: SAX

Como ya se ha indicado en el Capítulo 3, el escáner XML es el componente encargado de analizar la información contenida en un documento y transformarla en tokens, los cuales serán procesados por el algoritmo de análisis secuencialmente. A fin de evitar las complejidades del procesamiento desde cero de documentos XML, como ya se ha puesto de manifiesto también en dicho capítulo, es conveniente basar los analizadores en un marco de procesamiento XML genérico. Para ello, y siguiendo las directrices expuestas en el Capítulo 3, será necesario proporcionar un marco de integración adecuada. El objetivo es proporcionar un escáner XML, que permita al núcleo de análisis ver el marco genérico de procesamiento XML como un analizador léxico.

La Figura 5.3.37 esquematiza un posible marco de integración con SAX. De esta forma:

- La clase *Parser*SAX implementa la interfaz estándar de SAX *DefaultHandler*, realizando la conexión con el motor de procesamiento SAX mediante el establecimiento de los métodos de procesamiento de eventos apropiados, y realizando el mapeado de elementos XML a tokens. Por su parte, la clase *Scanner*SAX proporciona el scanner XML en sí, de cara al núcleo de análisis.

- Por su parte, la clase *Token* modeliza la visión de los elementos de información XML como tokens. Para determinar el código de token utiliza una tabla de mapeo modelizada por la clase *parsetable.model.Symbols*. También aloja la información sobre los atributos léxicos asociados (si dispone de ellos, como *text* de *#pcdata,* o atributos de etiquetas de apertura). Estos atributos léxicos se alojan mediante *AttributesImp*, clase que utiliza una tabla de tipo nombre de atributo-valor para alojar los atributos léxicos, y que implementa los métodos de acceso-establecimiento de atributos léxicos de la interfaz *AttributesI* (para el elemento XML #*pcdata*, se construye un token con un único atributo de nombre *text* y cuyo valor es el propio contenido textual que refiere el elemento).

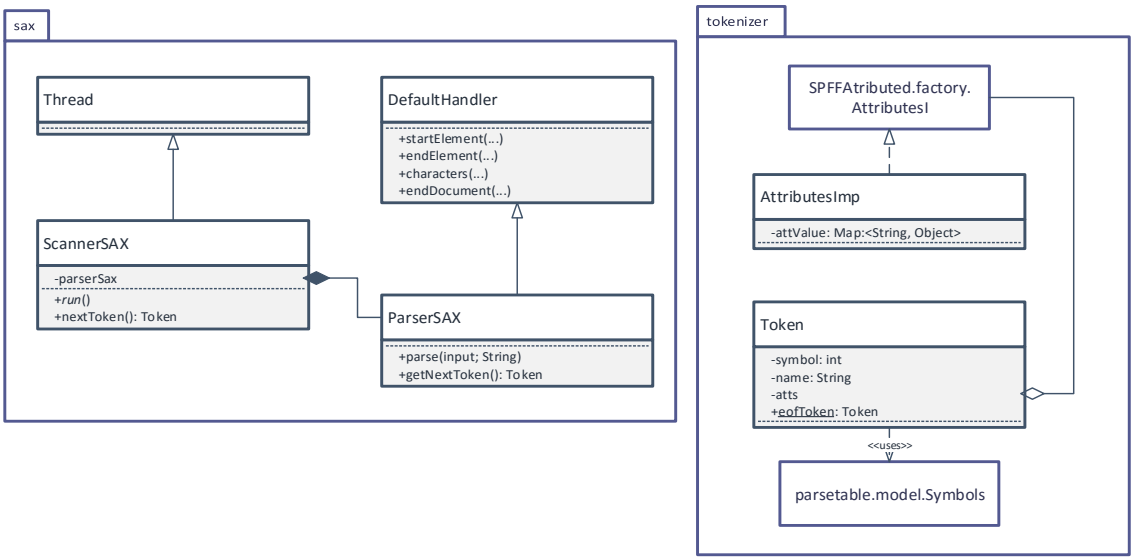


Figura 5.3.37. Clases para la configuración del analizador léxico SAX y construcción de tokens.

De esta forma, el documento XML se abrirá para el análisis mediante el método *parse* de *ParserSAX*. Cada token reconocido, disponible para consumir, se accederá y consumirá a través de su método *getNextToken*. Debido a que el algoritmo de análisis será el proceso principal que demande tokens secuenciales, por peticiones, a esta clase, ésta se ejecutará en un hilo independiente, gestión que realizará *ScannerSAX*, y a través de la cual, el algoritmo de análisis obtendrá dichos tokens, de manera indirecta, mediante su método *nextToken*. Así mismo, el inicio del proceso se realizará mediante el método *run* de *ScannerSAX*.

5.3.3.2 Construcción de los bosques de análisis atribuidos

El núcleo de análisis debe generar, como resultado, una representación atribuida del bosque de análisis sintáctico asociado con el documento analizado. Para ello, se extiende el modelo SPPF básico con campos para habilitar la subsecuente fase de evaluación semántica. El modelo resultante se denomina SPPFA (SPPF *atribuido*). Más concretamente, el modelo SPPF puede concebirse como un modelo de información constituido por dos tipos de nodos:

- Nodos terminales. Representan terminales y, por tanto, los nodos hoja del bosque.
- Nodos no terminales. Representan no terminales, y, por tanto, los nodos interiores del bosque. Estos nodos pueden tener varias listas de hijos. Presentarán, por tanto, empaquetamiento, cuando tengan dos o más listas de hijos. Una lista de hijos vacía representará el cuerpo de una producción nula (es decir, λ).

El modelo SPPFA surge de extender el modelo SPPF con campos para alojar atributos semánticos, así como con mecanismos para computar y acceder a sus valores. Debido a que el valor de un atributo puede no estar disponible desde el principio, estos campos destinados a

alojar dichos atributos y sus mecanismos de cómputo y acceso referencian a un objeto denominado *gestor del valor*. La Figura 5.3.38 refleja ambos modelos y sus diferencias.

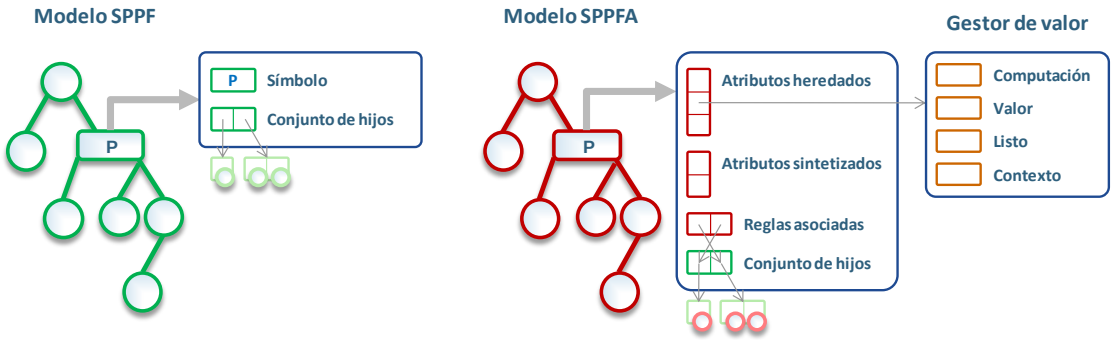


Figura 5.3.38. Modelos de nodos SPPF y de nodos SPPFA no terminales.

De esta manera, los nodos SPPFA se caracterizan por:

- Nodos SPPFA terminales. Extienden los nodos SPPF con una lista de atributos léxicos. Dichos atributos se componen de un gestor del valor para terminales que contiene únicamente un campo para el valor.

- Nodos SPPFA no terminales. Constan de una lista de atributos heredados, una lista de atributos sintetizados, y una estructura que asocia una regla de producción con una lista de nodos hijos de manera unívoca. La asociación que realiza esta estructura se caracteriza porque la lista de nodos hijos se corresponde con el cuerpo de la producción asociada. Los atributos sintetizados y heredados se componen de un gestor del valor para no terminales. Además de un campo para el valor, dichos gestores incluyen: (i) un proceso de cómputo (ecuación semántica codificada que computa el valor), (ii) un *flag* que indica si el valor está o no disponible, y (iii) un contexto, que representa la información requerida por el proceso de cómputo del valor (para el cómputo de los atributos sintetizados el contexto es el nodo actual, mientras que para el cómputo de los atributos heredados el contexto es el nodo padre).

La construcción del SPPFA se acopla con el núcleo de análisis a través de los desplazamientos y reducciones obrados en el mismo. Más concretamente:

- Las acciones de desplazamiento crean nodos terminales.

- Las acciones de reducción para producciones asociadas con no terminales que no son de núcleo crean los correspondientes nodos no terminales.

- Antes de comenzar las reducciones asociadas con un no terminal de núcleo se crea el correspondiente nodo para dicho no terminal. Dichas reducciones, por su parte, realizan el empaquetado de todas las listas de nodos hijo.

Dado que dicha construcción se realiza en sentido ascendente, desde las hojas hacia la raíz en el bosque, se garantiza que los nodos hijo, y toda la información asociada, siempre existirá a la hora de crear su respectivo nodo padre. Este hecho permite fijar el contexto de los gestores de valores. Efectivamente:

- Cuando se crea un nodo no terminal, dicho nodo puede fijarse como contexto de sus atributos sintetizados.
- Cada vez que se añade una lista de hijos a dicho nodo, puede inyectarse este nodo a los gestores de valores de los atributos heredados de cada uno de los hijos.

A fin de ejemplificar el proceso de construcción, desarrollaremos un ejemplo multivista sencillo para visualizar la construcción y estructura interna de los nodos de un bosque SPPFA. El ejemplo consistirá en dos tareas simples de procesamiento:

- Tarea 1. Obtener de un documento XML el valor textual que se presenta entre las etiquetas y .
- Tarea 2. Obtener de un documento XML el atributo *id* que se presenta entre las etiquetas y , y asociarlas al valor que se introduce a esta tarea como parámetro de entrada. Por último, imprimir por pantalla el atributo y valor asociado.

Para la tarea 1 se describe la vista **w**, y para la tarea 2 se describe la vista **u**. La Figura 5.3.39 muestra la gramática de atributos multivista para dichas tareas y la DTD y documento XML a procesar.

DTD

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT a (b) >
<!ELEMENT b (#PCDATA) >
<!ATTLIST b id CDATA #REQUIRED>
```

Instancia XML

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE a SYSTEM 'ab.dtd'>
<a>
  <b id = 'x'>7</b>
</a>
```

a) DTD (izquierda) y documento XML (derecha).

Vista u	Vista w
[p1] A ::= <a> B [e1] B.vh = B.v[w]	[p1] A ::= <a> B [e2] A.p = print(B.m[u])
[p2] B ::= C [e3] B. m = map(C.id, B.vh) [p3] C ::= #pcdata [e4] C.id = . id	[p4] B ::= #pcdata [e5] B.v = #pcdata. Text

b) Gramática de atributos multivista.

Figura 5.3.39. Ejemplo sencillo de gramática de atributos multivista (b) para el tipo de documentos mostrado en la parte superior (a).

La Figura 5.3.40 muestra el modelo de bosque SPPFA que se construye al procesar la instancia XML de la Figura 5.3.39. Para ello, el componente constructor utiliza la implementación específica de una factoría de nodos SPPFA que permite configurar el contenido de dichos nodos según la información aportada por la gramática de atributos multivista. Esta implementación puede ser generada de manera automatizada a partir de una especificación gramatical multivista, como se mostrará en la sección 5.3.3.3. Las etiquetas **pX** y **eX** de la Figura 5.3.40 se corresponden con las etiquetas **pX** moradas y **eX** verdes de la Figura 5.3.39. La etiqueta **pX** designa el identificador unívoco de una producción, y la etiqueta **eX** hace referencia a la ecuación semántica que se computa, cuya codificación del proceso de cómputo en sí se presenta en la sección 5.3.3.3.

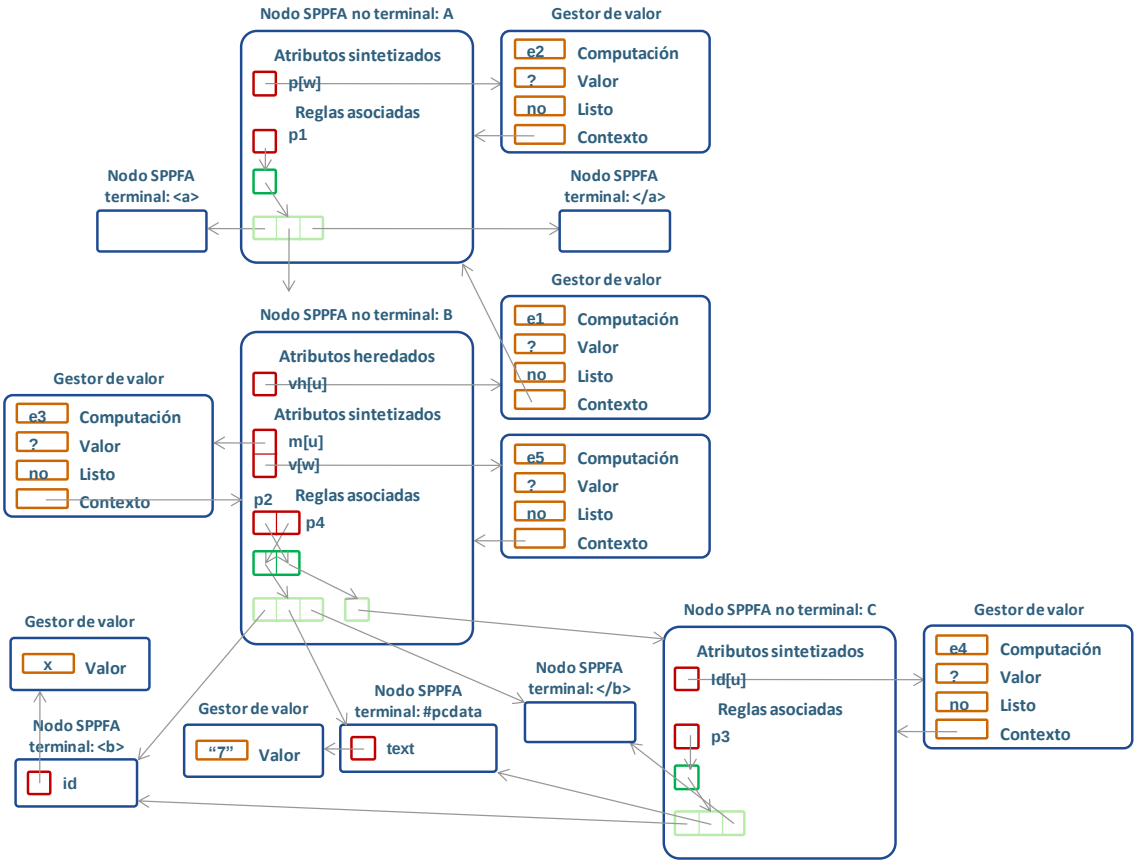


Figura 5.3.40. SPPFA del ejemplo sencillo construido en la fase de análisis.

Para finalizar, el diagrama de clases de la Figura 5.3.41 muestra una posible realización orientada a objetos del modelo SPPFA. De esta forma:

- Cada nodo SPPFA terminal se crea a partir del identificador *term* del símbolo desplazado, mediante la factoría de nodos que implementa *SPPFANodeFactoryI*, con el método *mkTNode(term: int, AttributesI atts).* En su construcción, se reserva espacio

para los atributos léxicos. A continuación, al nodo se le añade la información semántica. Los valores de los atributos sintetizados para este terminal se fijan en este momento, con gestores del valor para terminales, ya que los valores de dichos atributos ya se conocen y están contenidos en el símbolo desplazado.

- Cada nodo SPPFA no terminal se crea a partir del identificador *prod* de la regla de producción reducida, mediante la factoría de nodos que implementa *SPPFANodeFactoryI* con el método *mkNTNode(prod: int).* En su construcción, se reserva espacio para los atributos sintetizados y heredados. A dicho nodo se le asocia la lista de nodos hijos (nodos SPPFA) que conforma el cuerpo de la producción con el identificador de la regla de producción reducida. A continuación, al nodo se le añade la semántica. Para la computación de los valores de los atributos sintetizados de la cabeza de producción (el propio nodo SPPFA) se añaden gestores del valor para no terminales, y para los valores de los atributos heredados del cuerpo de la producción se añaden también gestores del valor para no terminales.

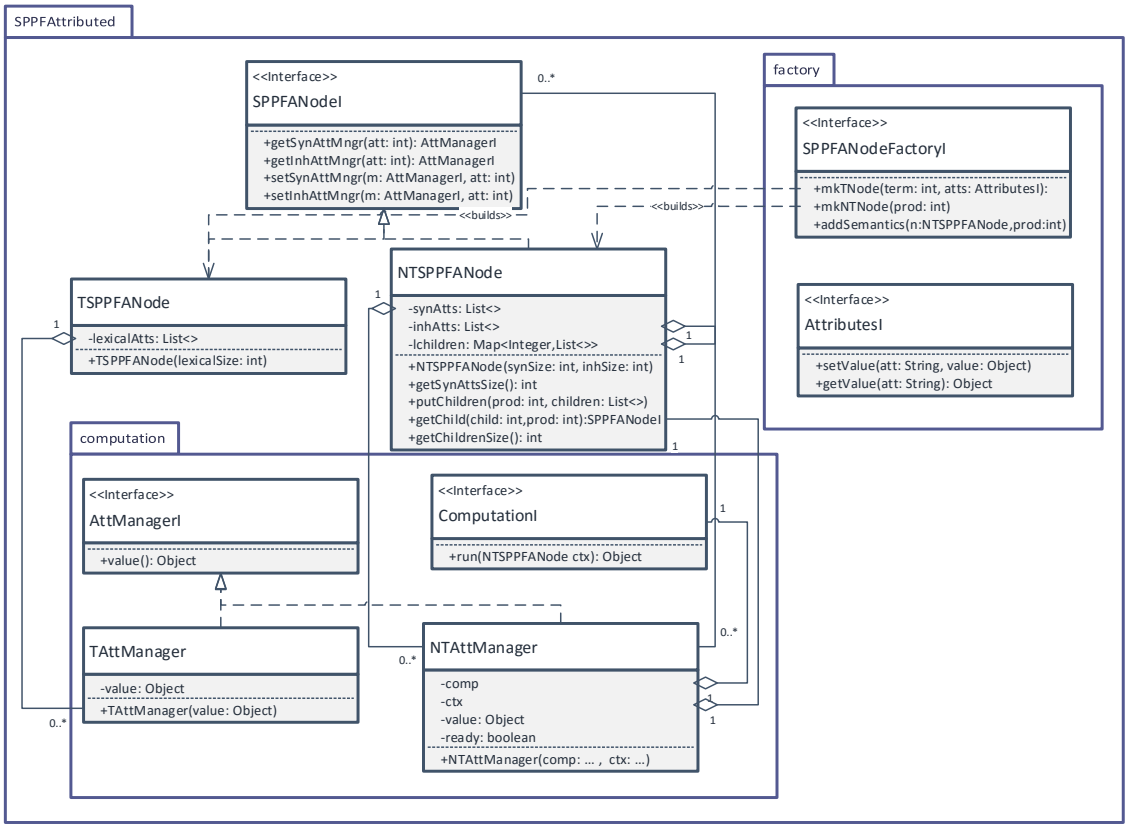


Figura 5.3.41. Diagrama de clases para una posible realización del modelo SPPFA.

5.3.3.3 La factoría de nodos SPPFA y su generación

La factoría de nodos es a lo que, en último término, se traduce la especificación gramatical multivista. Por tanto, la factoría de nodos que se utiliza es específica para la gramática de atributos multivista, y puede generarse automáticamente a partir de una descripción apropiada de la misma. En su generación, será necesario utilizar una *tabla de símbolos* y una *tabla de atributos* asociadas a la gramática, que sirvan para llevar a cabo la traducción de nombres simbólicos a identificadores unívocos numéricos utilizados en la implementación.

La Figura 5.3.42 muestra la plantilla de generación del código de la factoría de nodos SPPFA (la Figura 5.3.43 muestra el código generado para el ejemplo sencillo). Esta plantilla (Java), dará lugar a implementaciones de la factoría de nodos compuesta por los siguientes métodos:

- mkNTNode(int nt). Construye un nodo SPPFA no terminal en base a la selección dada por el identificador unívoco de dicho no terminal (nt) que figura en la tabla de símbolos. En su construcción, se fija el número de atributos sintetizados y heredados (tamaño de dichas listas) según la información contenida en la tabla de atributos final.

- mkTNode(int t, AttributesI atts). Construye un nodo SPPFA terminal en base a la selección dada por el identificador unívoco de dicho terminal (t) que figura en la tabla de símbolos. En su construcción, se fija el número de atributos sintetizados o léxicos (tamaño de dicha lista) según la información contenida en la tabla de atributos final. Adicionalmente, se fija el valor de tales atributos con gestores del valor para terminales. La información que se brinda a dichos gestores se busca en la tabla de atributos léxicos aportada por el token (atts) del propio terminal, (generado por el scanner XML). Gracias a la tabla de atributos, se calcula la posición exacta del atributo deseado en la lista de atributos del nodo.

- addSemantics(NTSPPFANode n, int prod). Funciona como un selector que, según el identificador de la regla de producción (prod) que consta en la tabla de símbolos, se asocia a cada uno de sus atributos sintetizados del nodo de entrada SPPFA un gestor del valor para nodos no terminales. Los atributos heredados de dicho nodo no se configuran aquí, sino que se realiza desde su nodo padre. Por ello, en este lugar, se realizan las configuraciones de los atributos heredados de los nodos hijos del nodo que correspondan. Gracias al identificador de producción (prod) y a las estructuras semánticas, se puede calcular la posición exacta del nodo hijo referido (en la lista de nodos hijos del nodo actual). Y mediante la tabla de atributos, se puede calcular la posición exacta del atributo deseado en la lista de atributos del nodo. Por último, los gestores del valor tendrán asociados un cómputo determinado (comp[i]), que es conocido, construido y almacenado desde el constructor de la factoría.

- Método constructor. Crea cada proceso de cómputo asociado a cada ecuación semántica existente en la gramática y lo almacena en una lista con una posición específica (más detalles en sección 5.4.1).

```
public class NodeFactoryImp implements SPPFANodeFactoryI {
  private SemanticClassInterface semObj;
  private ComputationI[] comp;

  public NodeFactoryImp () {
    semObj = new <ClaseSemántica>();
    comp = new ComputationI[<NúmeroExpresionesSemánticas>];

    comp[<NúmeroExpresiónSemántica>] = new ComputationI() {
      @Override
      public Object run(NTSPPFANode ctx) {
        return <ExpresiónSemántica>;
      }
    };
    …
    comp[<NúmeroExpresionesSemánticas - 1>] = …
  }

<ExpresiónSemántica>  → <Arg> | semObj.<NombreFunción>(<Arg>? (,<Arg>)*)
<Arg> → (<AtributoSintetizado> | <AtributoHeredado> | <ExpresiónSemántica>)
<AtributoSintetizado> → ctx.getChild(<Producción>,<Hijo>).getSynAttMngr(<Atributo>).value()
<AtributoHeredado> → ctx.getInhAttMngr (<Atributo>).value()

  public NTSPPFANode mkNTNode(int nt){
    switch(nt) {
      case <NoTerminal>: return new NTSPPFANode(<NúmeroAtributosSintetizados>,<NúmeroAtributosHeredados>);
      …
    }
    return null;
  }

  public TSPPFANode mkTNode(int t, Attributes atts) {
    TSPPFANode n = null;
    switch(t) {
      case <Terminal>: n = new TSPPFANode(<NúmeroAtributosLéxicos>); <MutadorLéxico>+ ; break;
      …
    }
    return n;
  }

<MutadorLéxico> → n.setSynAttMngr (new TAttManager(atts.getValue("<NombreAtributoLéxico>")),<AtributoLéxico>);

  public void addSemantics(NTSPPFANode n, int prod) {
    switch(prod) {
      case <Producción>: (<MutadorAtributoSintetizado>|<MutadorAtributoHeredado>)+; break;
      …
    }
  }
}

<MutadorAtributoSintetizado> → n.setSynAttMngr(new NTAttManager(comp[<NúmeroExpresiónSemántica>],n),
                                  <AtributoSintetizado>);
<MutadorAtributoHeredado> → n.getChild(prod,<Hijo>).
                        setInhAttMngr(new NTAttManager(comp[<NúmeroExpresiónSemántica>],n),<AtributoHeredado>);
```

Figura 5.3.42. Plantilla de generación de código de la factoría de nodos SPPFA.

NodeFactoryImp.java

```
private SemanticClassInterface semObj;
private ComputationI[] comp;
public NodeFactoryImp() {
  semObj = new SemanticClass();
  comp = new ComputationI[5];
  comp[0] = new ComputationI() {      @Override
    public Object run(NTSPPFANode ctx) { return semObj.print(ctx.getChild(0,1).getSynAttMngr(1).value()/*B.m[u]*/); [e2] }
  };
  comp[1] = new ComputationI() {      @Override
    public Object run(NTSPPFANode ctx) { return ctx.getChild(0,1).getSynAttMngr(0).value()/*B.v[w]*/; [e1] }
  };
  comp[2] = new ComputationI() {      @Override
    public Object run(NTSPPFANode ctx) { return semObj.map(ctx.getChild(1,0).getSynAttMngr(0).value()/*[u*0]C.id[u]*/,
                                                           ctx.getInhAttMngr(0).value()/*B.vh[u]*/); [e3] }
  };
  comp[3] = new ComputationI() {      @Override
    public Object run(NTSPPFANode ctx) { return ctx.getChild(2,1).getSynAttMngr(0).value()/*#pcdata.text*/; [e5] }
  };
  comp[4] = new ComputationI() {      @Override
    public Object run(NTSPPFANode ctx) { return ctx.getChild(3,0).getSynAttMngr(0).value()/*<b>.id*/; [e4] }
  };

@Override
public NTSPPFANode mkNTNode(int nt) {
  switch(nt) {
    case 0: return new NTSPPFANode(1,0); //A
    case 1: return new NTSPPFANode(2,1); //B
    case 2: return new NTSPPFANode(1,0); //[u*0]C
  }
  return null;
}

@Override
public TSPPFANode mkTNode(int t, AttributesI atts) {
  TSPPFANode n = null;
  switch(t) {
        case 4: n = new TSPPFANode(0); //</b> break;
        case 5: n = new TSPPFANode(0); //$      break;
        case 1: n = new TSPPFANode(1); //<b>  n.setSynAttMngr(new TAttManager(atts.getValue("id")),0);  break;
        case 3: n = new TSPPFANode(0); //<a>  break;
        case 0: n = new TSPPFANode(1); //#pcdata n.setSynAttMngr(new TAttManager(atts.getValue("text")),0);  break;
        case 2: n = new TSPPFANode(0); //</a> break;
  }
  return n;
}

@Override
public void addSemantics(NTSPPFANode n, int prod) {
  switch(prod) {
    case 0://A ::= <a> B </a>
        n.setSynAttMngr(new NTAttManager(comp[0],n),0)/*A.p[w]*/;
        n.getChild(prod,1).setInhAttMngr(new NTAttManager(comp[1],n),0)/*B.vh[u]*/;   break;
    case 1://B ::= [u*0]C
        n.setSynAttMngr(new NTAttManager(comp[2],n),1)/*B.m[u]*/;   break;
    case 2://B ::= <b> #pcdata </b>
        n.setSynAttMngr(new NTAttManager(comp[3],n),0)/*B.v[w]*/;    break;
    case 3://[u*0]C ::= <b> #pcdata </b>
        n.setSynAttMngr(new NTAttManager(comp[4],n),0)/*[u*0]C.id[u]*/;   break;
  }
 }
}
```

Figura 5.3.43. Factoría de nodos SPPFA para el ejemplo sencillo.

5.4 La fase de evaluación

Una vez finalizada la fase de análisis, se habrá producido una representación atribuida apropiada del bosque de análisis sintáctico (en forma de una instancia del modelo SPPFA introducido en la sección anterior). Sobre este bosque atribuido podrá aplicarse un método de evaluación, adaptando alguna de las estrategias de evaluación de árboles de análisis sintáctico ya existentes (p.e., dirigidas por los datos o bajo demanda). Más concretamente, en esta tesis hemos adoptado una estrategia de evaluación bajo demanda, análoga a la utilizada en patrones para la evaluación de atributos en árboles atribuidos como los descritos en [Magnusson & Hedin 2007]. Esta estrategia consiste en que los atributos de los distintos nodos del bosque se computan *bajo demand*a. En otras palabras, los valores de los atributos de los nodos del bosque se calculan de manera única la primera vez que se requieren y quedarán almacenados para futuras consultas. El cálculo de dichos valores se realizará a través de procedimientos de cómputo que codifican las ecuaciones semánticas, y será controlado por los ya citados gestores del valor. Este aspecto se detalla en la sección 5.4.1. El proceso de evaluación del bosque se presenta en la sección 5.4.2.

5.4.1 Gestores del valor, cómputo semántico y clase semántica

Como ya se ha indicado en la sección anterior, los gestores del valor son gestores asociados a atributos sintetizados o heredados que definen su valor. A través de cada gestor se almacena, se accede o se computa el valor del correspondiente atributo. Como también se ha indicado ya, los gestores del valor pueden ser de dos tipos: para terminales y para no terminales.

```
public interface AttManagerI {
  public Object value();
}
```

```
public class NTAttManager implements AttManagerI {
  private ComputationI comp;
  private Object value;
  private boolean ready;
  private NTSPPFANode ctx;
  public NTAttManager(ComputationI comp, NTSPPFANode ctx) {
    this.comp = comp;
    this.ctx = ctx;
    this.ready = false;
  }
  @Override
  public Object value() {
    if (!ready) {
      value = comp.run(ctx);
      ready = true;
    }
    return value;
  }
}
```

```
public class TAttManager implements AttManagerI {
  private Object value;
  public TAttManager(Object value) {
    this.value = value;
  }
  @Override
  public Object value() {
    return value;
  }
}
```

Figura 5.4.1. Implementación del gestor del valor para nodos no terminales (izquierda) y terminales (derecha).

La Figura 5.4.1 muestra la codificación de los dos tipos de gestores del valor: para nodos terminales con la clase *TAttManager*, y para nodos no terminales con la clase *NTAttManager*, ambas implementando la interfaz *AttManagerI* de gestor de valor genérico (véase diagrama de clases en Figura 5.3.41). Su contenedor del valor es *value*, y su campo de computación (únicamente para nodos no terminales) es *comp*. Efectivamente, y tal y como se ha indicado también ya en la sección anterior:

- El gestor del valor para terminales es simplemente un contenedor de un valor preestablecido en el momento de construcción del gestor.
- Sin embargo, el gestor del valor para no terminales posee el campo que aloja el valor del atributo como vacío o desconocido en su construcción. Cuando a dicho gestor se le solicita el valor por primera vez, recurre al campo adicional de computación para calcular dicho valor. Este proceso es el proceso de cómputo del valor del atributo descrito por la correspondiente ecuación semántica de la gramática de atributos multivista.

Para realizar las computaciones requeridas por los gestores del valor para nodos no terminales, previamente cada ecuación semántica se codifica mediante una implementación de la interfaz *ComputationI* (sección 5.3.3.3), que, de este modo, proporciona el correspondiente *procedimiento de cómputo*. De esta forma, los campos de computación en los gestores de valor albergarán una referencia al objeto de tipo *ComputationI* apropiado. La ejecución de estos procedimientos se lleva a cabo a través del método *run* introducido por la interfaz *ComputationI* (p.e., en el ejemplo simple dichos métodos ejecutarán las implementaciones de los procedimientos **eX** concretos resaltados en la Figura 5.3.43 respecto a la Figura 5.3.39).

Cada procedimiento de cómputo, y por tanto el método *run* de *ComputationI*, necesita recibir un contexto (*ctx*), un nodo SPPFA de referencia, a través del cual poder buscar y acceder a los atributos que constan en ambas partes (derecha e izquierda) de la ecuación semántica. Dicho contexto se almacena en el gestor del valor como ya se ha descrito en la sección anterior. Así mismo, cada atributo semántico se representa mediante un código numérico único. Con ello, se puede almacenar y acceder de manera precisa al atributo de un nodo SPPFA (métodos de *SPPFANodeI*, Figura 5.3.41) utilizando dichos códigos. Establecido el nodo de contexto SPPFA (*ctx*), nunca será necesario acceder al nodo padre de dicho contexto, pero sí a sus nodos hijos (el nodo contiene una lista de nodos hijos por cada producción distinta que alberga). Para ello, también existe una asociación unívoca de cada regla de producción con un identificador numérico. De esta manera, se puede acceder de forma precisa a un nodo perteneciente a un cuerpo de producción, de una de las vistas, mediante el identificador numérico de la regla de producción y la traducción de la posición del elemento en el cuerpo de la producción (método *getChild* de *NTSPPFANode*, Figura 5.3.41). Una vez calculado el valor del atributo, el gestor del valor lo almacena y actualiza su indicador (*ready*) para que en futuras peticiones se acceda directamente al valor almacenado.

SemanticClass.java

```
import java.util.HashMap;
public class SemanticClass implements SemanticClassInterface {
  @Override
  public Object print(Object m) {
    System.out.println(m);
    return null;
  }
  @Override
  public Object map(Object id, Object v) {
    HashMap m = new HashMap();
    m.put(id, v);
    return m;
  }
}
```

SemanticClassInterface.java

```
public interface SemanticClassInterface {
  public Object print(Object a0);
  public Object map(Object a0, Object a1);
}
```

Figura 5.4.2. Clase Semántica del ejemplo sencillo y respectiva Interfaz de Clase Semántica.

Como puede observarse en la Figura 5.3.43, los procedimientos de cómputo hacen referencia a métodos externos, como *print* y *map*. Estos métodos deben implementarse en una *clase semántica*, clase que no podrá generarse de forma automática, aunque sí podrá generarse una interfaz *SemanticClassInterface* que indique con exactitud qué métodos deberá implementar dicha clase semántica. La Figura 5.4.2 muestra la clase semántica del ejemplo sencillo y la interfaz generable de manera automática.

5.4.2 Evaluación en el modelo SPPFA

El mecanismo de evaluación por demanda del modelo SPPFA reside en la manera de computar los valores de cada atributo asociado a los nodos del bosque. Comienza al realizarse la petición de los valores de los atributos sintetizados del nodo raíz del bosque. Concretamente, éste cálculo se realiza a través del gestor del valor de un atributo, lo que dispara un proceso de evaluación por demanda caracterizado por:

- Cuando se solicita el valor de un atributo, dicha solicitud se delega en el gestor del valor de dicho atributo.
- Si el gestor conoce el valor, lo devuelve directamente.
- En otro caso, el gestor ejecuta el procedimiento de cómputo asociado. Calculado el valor, se almacena para futuras peticiones.

Puesto que los procedimientos de cómputo evalúan la expresión funcional de la parte derecha de la ecuación semántica que codifican (Figura 5.3.43 respecto a la Figura 5.3.39), el orden de evaluación quedará definido como:

- La evaluación de una función $f(a_0, \ldots, a_k)$ se realiza solicitando/evaluando en orden de izquierda a derecha cada uno de sus argumentos, y finalmente se evalúa la función f definida sobre dicha lista de argumentos.
- La evaluación de una referencia a un atributo se traduce en solicitar el valor de dicho atributo.
- La evaluación de un valor prefijado se traduce en devolver dicho valor como tipo cadena.

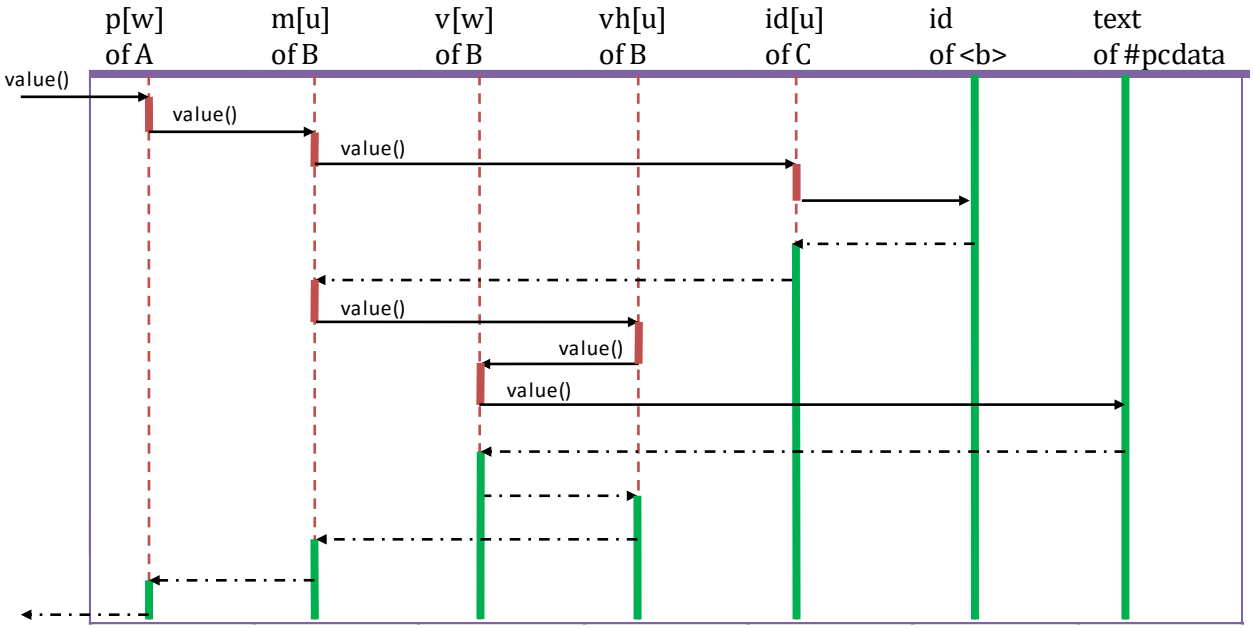


Figura 5.4.3. Proceso de evaluación del SPPFA de la Figura 5.3.40.

La Figura 5.4.3 muestra en un diagrama de secuencias el proceso de evaluación del bosque SPPFA de la Figura 5.3.40, correspondiente al ejemplo sencillo que estamos utilizando. La evaluación comienza con la petición del atributo sintetizado *p[w]* de la raíz del bosque. El eje vertical simula el tiempo de ejecución, incremental de arriba abajo. Las barras verticales en rojo muestran el momento en el que se realiza una petición del atributo descrito en la parte superior, cuyo valor es desconocido. Las flechas etiquetadas con *value* indican la petición del valor del atributo correspondiente en su eje vertical, ejecutando dicho método en su gestor del valor asignado. Como ya se ha indicado, este valor se establece, para atributos de símbolos no terminales, mediante la ejecución del procedimiento de cómputo del gestor del valor de dicho atributo, que se inicia bajo su método *run*. Las flechas punteadas indican el establecimiento y retorno de un valor. Las barras verticales en verde muestran el momento en que el valor del atributo descrito en la parte superior ya se ha calculado y, por tanto, está disponible.

5.5 XLOP3

XLOP3 es un entorno para el desarrollo de aplicaciones XML con gramáticas de atributos multivista. Este entorno ha sido concebido y construido durante este trabajo de tesis para demostrar la viabilidad real de las gramáticas de atributos multivista para la generación de aplicaciones de procesamiento de documentos XML. El nombre del entorno XLOP3 atiende al entorno XLOP descrito en el Capítulo 3. Si bien la arquitectura interna de XLOP3 es por completo diferente a la de XLOP, XLOP3 hereda la idea conceptual de éste último: generar aplicaciones de procesamiento XML a partir de especificaciones declarativas de alto nivel. Así mismo, XLOP3 pone especial énfasis en resolver las limitaciones identificadas en nuestros trabajos previos sobre desarrollo de aplicaciones XML dirigido por lenguajes: conformidad de las gramáticas específicas de procesamiento con respecto a las gramáticas documentales, y soporte bien fundamentado a la modularidad de las especificaciones.

En esta sección se describe XLOP3. En la sección 5.5.1 se presenta la visión general del entorno XLOP3. La sección 5.5.2 detalla el lenguaje de especificación de XLOP3 para la descripción declarativa de tareas de procesamiento de documentos XML mediante gramáticas de atributos multivista. La sección 5.5.3 muestra cómo se configura y funciona el entorno. XLOP3. La sección 5.5.4 describe el entorno de ejecución y depuración. La sección 5.5.5 analiza, por último, la arquitectura interna del sistema, mostrando cómo se plasma el modelo de las gramáticas de atributos multivista y su realización operacional.

5.5.1 Visión general

XLOP3 (XML *Language-Oriented Processing* 3.0) es un entorno para el desarrollo, mantenimiento y evolución de aplicaciones de procesamiento XML basados en gramáticas de atributos multivista. Los procesadores de documentos generados por XLOP3 permiten, mediante un único análisis de los documentos XML, la realización de múltiples tareas de procesamiento. Estas tareas se describen mediante especificaciones gramaticales multivista que, a diferencia de su predecesor XLOP, donde las especificaciones se transforman en implementaciones CUP, configuran el motor de traducción genérico integrado en el entorno de ejecución y depuración aportado por XLOP3 como componente principal de las aplicaciones de procesamiento XML generadas. Los principales elementos de los que se componen estas aplicaciones se muestran en el proceso de desarrollo de una aplicación de procesamiento XML con XLOP3, en la Figura 5.5.1.

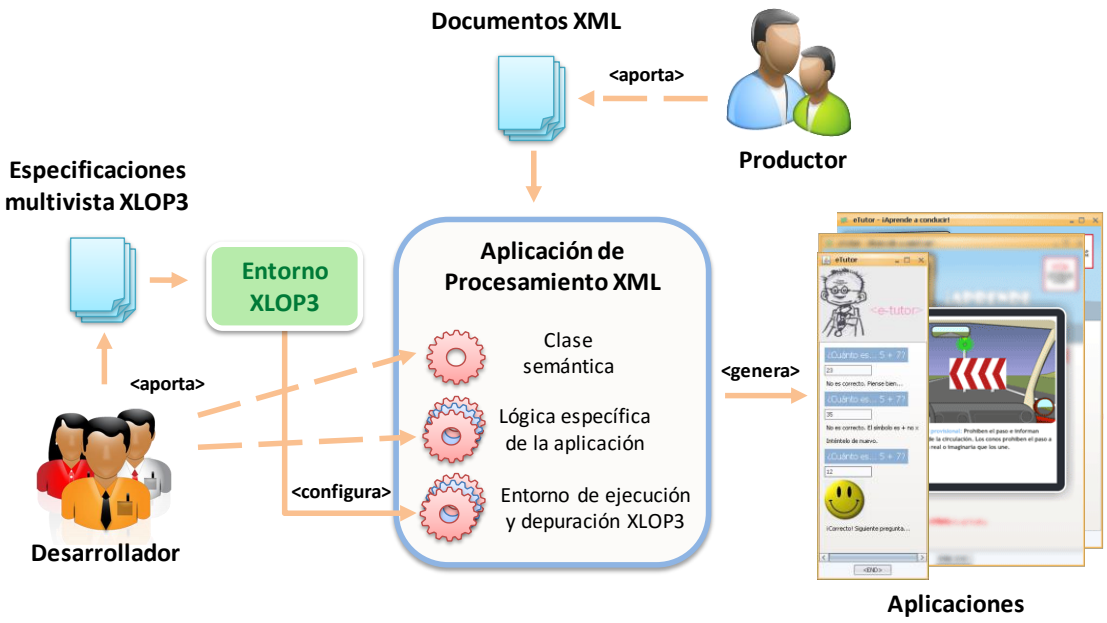


Figura 5.5.1. Desarrollo de aplicaciones de procesamiento XML con XLOP3.

El desarrollo de aplicaciones de procesamiento XML con XLOP3 se caracteriza por los siguientes elementos y actividades:

- Los desarrolladores aportan las especificaciones multivista que describen las tareas de procesamiento XML que serán llevadas a cabo por la aplicación, con una gramática de atributos multivista. Para dicho propósito, se utiliza el lenguaje de especificación de XLOP3.

- Las especificaciones multivista son procesadas por el entorno XLOP3, obteniendo una configuración para el entorno de ejecución y depuración aportado por el mismo entorno XLOP3. Este último incorpora un traductor específico, con soporte para gramáticas de atributos multivista, que funcionará como componente principal que dirigirá el procesamiento de los documentos XML.

- El traductor configurado requerirá una implementación de las funciones semánticas utilizadas en la especificación XLOP3. Los desarrolladores deben aportar estas funciones como métodos de una clase semántica. Esto permite conectar la capa de la lógica específica de la aplicación con la capa lingüística del proceso de traducción.

- Las funciones semánticas utilizadas en las especificaciones XLOP3 podrán utilizar un marco específico para la aplicación, la lógica específica de la aplicación, que también deberá ser proporcionado por los desarrolladores. Al igual que en otros de nuestros enfoques al procesamiento de documentos XML dirigido por lenguajes (Capítulo 3), la interconexión se lleva a cabo a través de la implementación de las funciones semánticas como métodos de la clase semántica.

- La aplicación de procesamiento XML quedará constituida por la clase semántica, la lógica específica de aplicación y el entorno de ejecución y depuración de XLOP3 configurado por una serie de componentes generados por el entorno XLOP3.
- Finalmente, los productores de recursos podrán aportar documentos XML a la aplicación de procesamiento XML para generar instancias de aplicaciones XML a partir de la información presente en dichos documentos.

Para que este proceso pueda realizarse correctamente, el entorno XLOP3 debe ser adecuadamente configurado en función de la aplicación específica. En las siguientes secciones se detalla cómo se realiza la generación de una aplicación de ejemplo (caso de ejemplo de las fórmulas aritméticas, sección 5.2.1) utilizando XLOP3.

5.5.2 Lenguaje de especificación

XLOP3 permite describir, por una parte, gramáticas EBNF de base, y, por otra, vistas gramaticales. Una vista gramatical se especifica en XLOP3 de manera modular, mediante subgramáticas. Una subgramática es una porción de una vista gramatical, en donde su sintaxis se describe con una estructura gramatical (bloque *Structure*), y su semántica, con una semántica gramatical (bloque *Semantics*). La sección 5.5.2.1 describe la notación XLOP3 para la especificación de gramáticas EBNF de base. En la sección 5.5.2.2 se presenta el lenguaje de especificación de XLOP3 para estructuras gramaticales, y cómo éstas deben definirse para ser correctas en el sentido de "estructura bien formada". De la misma manera, se presenta en la sección 5.5.2.3, el lenguaje de especificación XLOP3 para semánticas gramaticales, y cómo deben definirse para que las extensiones semánticas de las estructuras gramaticales que realizan estén "bien formadas".

5.5.2.1 Gramáticas EBNF de base

En XLOP3 las gramáticas EBNF de base se especifican de similar manera a la notación utilizada en el Capítulo 4. En concreto, una gramática EBNF se compone de una o más reglas de producción. Estas reglas seguirán la siguiente notación (los corchetes indican el tipo de información que debe sustituirse en su lugar):

- Regla de producción:

  [cabeza de producción] **::=** [expresión regular]**;**

Las expresiones regulares poseen como símbolos del alfabeto, etiquetas XML de apertura, etiquetas XML de cierre (pero no admite la notación simplificada de etiqueta de apertura y cierre), o terminales (p. e., los símbolos de núcleo son terminales en este caso). Por otra parte, la notación de una expresión regular (véase Capítulo 4, sección 4.3.1) es, considerándose α y β expresiones regulares, de la siguiente manera:

- Unión de α+β: α **|** β
- Concatenación de α·β: α β
- Agrupación de α: **(**α**)**
- Iteración de cero o más veces α: α*****
- Iteración de una o más veces α: α**+**
- Una o cero veces α: α**?**

Por último, hay que señalar que se considera como axioma de la gramática el símbolo cabeza de producción de la primera regla que aparezca en la especificación. Obsérvese también que esta notación es la misma que utiliza la herramienta *Grammar Equivalence Checker* (Capítulo 4), salvo la peculiaridad de que en dicha herramienta se permite especificar terminales con caracteres especiales entre comillas simples, mientras que en XLOP3 esto no es necesario, pues los únicos símbolos terminales que contempla son etiquetas XML, o *#pcdata*.

5.5.2.2 Estructuras gramaticales

En XLOP3 las subgramáticas que realizan símbolos *de núcleo* de la gramática EBNF base se denominan *estructuras gramaticales*. Dichas estructuras se describen mediante bloques *Structure*, construcción que permite que varias vistas compartan, de manera efectiva, la misma subgramática para un determinado símbolo de núcleo, ya que, al describir tales bloques, es posible indicar la vista o vistas a los que pertenecen. Con ello se minimiza la redundancia de las especificaciones, y se enfatiza la compartición estructural a nivel de construcción y evaluación de los árboles de derivación. Una estructura gramatical, para una o más vistas gramaticales, se especifica en XLOP3 de acuerdo con los siguientes convenios de notación (los caracteres y palabras en **negrita** son reservados):

- Estructura gramatical:

  Structure of [símbolo de núcleo] **in** [nombres de vista separados por ','] **{**
  [una o más reglas de producción]
  }

- Regla de producción:

  [cabeza de producción] **::=** [secuencia de símbolos terminales/no terminales/de núcleo]**;**

Para el soporte XML, en XLOP3 los no terminales son cadenas de caracteres que no empiezan por números ni contienen caracteres especiales. Los terminales son etiquetas XML de apertura (**<**[nombre etiqueta]**>**) o cierre (**</**[nombre etiqueta]**>**) y se escriben de la misma manera que en las instancias XML. No se admite la notación simplificada de etiqueta de apertura y cierre (**<**[nombre etiqueta]**/>**), sino que se obliga a que se descomponga en su

respectiva etiqueta de apertura y su respectiva etiqueta de cierre, a fin de evitar confusiones durante el desarrollo y depuración de las gramáticas, y en el momento de depurar el procesamiento de los documentos XML. Por otra parte, el contenido textual XML, que se escribe mediante la palabra reservada *#pcdata*, se considera también un terminal. Los cuerpos de las producciones nulas, que hemos representado mediante λ, en XLOP3 no tienen representación simbólica: simplemente se omite el cuerpo de la producción correspondiente.

La notación descriptiva expuesta garantiza especificar estructuras gramaticales sintácticamente correctas, pero no bien formadas. Para que una estructura gramatical esté bien formada, ésta debe cumplir los siguientes requisitos:

- Toda regla de producción debe ser alcanzable y no existir duplicada.
- Los nombres de los símbolos no terminales/terminales/de núcleo han de ser disjuntos.
- No se puede utilizar, como símbolo cabeza de producción, otro símbolo de núcleo distinto al axioma.
- La estructura debe tratarse de una gramática no autoembebible.
- El lenguaje que describe debe ser equivalente al descrito por la expresión regular de la regla EBNF asociada al no terminal de núcleo, si la gramática EBNF de base está disponible, o al descrito por el resto de las estructuras para el no terminal de núcleo si tal gramática EBNF no se ha proporcionado.

Así mismo, es necesario considerar también las siguientes restricciones globales a nivel de vista:

- Para cada no terminal de núcleo y para cada vista, existe exactamente una estructura gramatical que asigna la subgramática para dicho no terminal de núcleo con dicha vista.
- La gramática incontextual asociada a la vista debe ser no ambigua, a fin de que cada vista imponga un único árbol de derivación sobre cada sentencia analizada. En XLOP3 dicha condición se substituye por la condición más fuerte de que la gramática sea una gramática LALR(1).

Es importante, por último, señalar que todos los no terminales de núcleo referidos en una estructura gramatical son locales a dicha estructura gramatical. De esta forma, en otras estructuras gramaticales podrán utilizarse los mismos nombres de no terminales, pero se tratarán, en cualquier caso, de símbolos diferentes.

La Figura 5.5.2 presenta las especificaciones XLOP3 para las estructuras gramaticales de las fórmulas aritméticas respecto a la gramática XLOP3 EBNF de base, y todo ello respecto a la gramática multivista elaborada en la sección 5.2.3.

<table>
<tr><th>Gramática EBNF de base</th><th>Vista idr y env</th><th>Vista eva</th></tr>
<tr><td>f -> <f> exp </f>;</td><td colspan="2">Structure of f in idr, env, eva {
f ::= <f> exp </f> ;
}</td></tr>
<tr><td>exp -> opnd ((a|m) opnd)*;</td><td>Structure of exp in idr, env {
exp ::= opnd rexp ;
rexp ::= op opnd rexp ;
rexp ::= ;
op ::= a ;
op ::= m ;
}</td><td>Structure of exp in eva {
exp ::= expr ;
expr ::= expr a term ;
expr ::= term ;
term ::= term m opnd ;
term ::= opnd ;
}</td></tr>
<tr><td>opnd -> n|v|p;</td><td colspan="2">Structure of opnd in idr, env, eva {
opnd ::= n ;
opnd ::= v ;
opnd ::= p ;
}</td></tr>
<tr><td>n -> <Num> #pcdata </Num>;</td><td colspan="2">Structure of n in idr, env, eva {
n ::= <Num> #pcdata </Num> ;
}</td></tr>
<tr><td>v -> <Var> #pcdata </Var>;</td><td colspan="2">Structure of v in idr, env, eva {
v ::= <Var> #pcdata </Var> ;
}</td></tr>
<tr><td>p -> exp ;</td><td colspan="2">Structure of p in idr, env, eva {
p ::= exp ;
}</td></tr>
<tr><td>a -> <Sum> </Sum>;</td><td colspan="2">Structure of a in idr, env, eva {
a ::= <Sum> </Sum> ;
}</td></tr>
<tr><td>m -> <Mul> </Mul>;</td><td colspan="2">Structure of m in idr, env, eva {
m ::= <Mul> </Mul> ;
}</td></tr>
</table>

Figura 5.5.2. Estructuras gramaticales en XLOP3 de las vistas idr, env y eva, y correspondencia respecto a la gramática EBNF de base de las fórmulas aritméticas.

5.5.2.3 Semánticas gramaticales

Una semántica gramatical es la porción de extensión semántica de una estructura gramatical, para una única vista gramatical. De esta forma, en el caso de que la estructura gramatical se comparta con otras vistas, será necesario especificar semánticas diferentes para vistas diferentes. Puesto que, en una gramática de atributos, la semántica está ligada a la gramática incontextual, en las descripciones de una semántica gramatical deben aparecer las mismas reglas de producción que aparecen en la respectiva estructura gramatical. Una semántica gramatical se especifica en XLOP3 de la siguiente manera:

- Semántica gramatical:

  Semantics for [símbolo de núcleo] **of** [nombre de vista] **{**
  [una o más reglas de producción con semántica]
  }

- Regla de producción con semántica:

  [regla de producción (la misma definida en su estructura gramatical)] **{**
  [cero o más ecuaciones semánticas]
  }

- Ecuación semántica: [referencia atributo] **=** [referencia atributo/valor/función semántica]**;**

- Referencia a atributo (el contenido entre caracteres '#' es opcional):

  [atributo] #nombre de vista entre '**[**' y '**]**'# **of** [terminal/no terminal] #número de ocurrencia entre '**(**' y '**)**'#

- Los valores se especifican entre comillas dobles '**""**' y su tipo por defecto es el tipo cadena. Las funciones semánticas, cuyos nombres y número de parámetros refieren a los métodos definidos en la clase semántica (sección 5.4.1), son: [nombre de función] **(**[lista de referencia atributo/valor/función semántica separados por '**,**']**)**

El número de ocurrencia puede omitirse si es cero. Bajo este formalismo, una ecuación semántica escrita tradicionalmente como: **A.h = f(B(1).v, B.v)** para una producción **A ::= B B**, se escribe como**: h of A = f(v of B(1), v of B)**. Cuando un atributo pertenece a otra vista (este caso sólo es posible si el atributo refiere a un símbolo de núcleo), el atributo debe escribirse acompañado del nombre de dicha vista entre corchetes. Si **B** hubiera sido un símbolo de núcleo, entonces es necesario indicar con exactitud de qué vista procede su atributo **v**, aportando entre corchetes el nombre de tal vista como sufijo del nombre del atributo semántico. Por ejemplo, si para **B** existiese en una vista **vista1** un atributo sintetizado **v**, y otro atributo **v** en una segunda vista **vista2**, dichos atributos podrían referirse en una ecuación como, por ejemplo: **h of A = f(v[vista1] of B(1), v[vista2] of B)**. No es necesario declarar esta referencia de vista explícitamente cuando los atributos del símbolo de núcleo referidos pertenecen a la vista que se está describiendo, ya que, por defecto, se deduce que tales atributos pertenecen a la vista actual que se describe (se denominarán *atributos locales a la vista*, frente a los de las otras vistas, que se denominarán *externos*). Por último, las funciones semánticas son referencias a invocaciones de métodos cuya definición e implementación, en código Java, debe aportarse a través de la clase semántica (sección 5.5.3.1).

Para que una semántica gramatical esté bien formada, ésta debe cumplir con una serie de requisitos o restricciones que dependen del contexto. Este contexto se establece en función de los atributos semánticos que se declaran para una vista, una estructura gramatical y un símbolo. Cada símbolo no terminal/de núcleo de una semántica gramatical puede poseer un conjunto de atributos sintetizados y un conjunto de atributos heredados propio. Ambos conjuntos deben ser disjuntos entre sí. Por su parte, los símbolos terminales sólo pueden

poseer un conjunto de atributos sintetizados, denominados, como es usual, atributos *léxicos*, cuyos valores se obtienen del análisis de los documentos.

En XLOP3, al igual que en XLOP, no es necesario declarar explícitamente cuáles son los atributos sintetizados y heredados de un símbolo, sino que estos se deducen de las producciones y ecuaciones semánticas. Concretamente, en una ecuación semántica de una producción, un atributo será sintetizado cuando:

- El atributo aparece en el lado izquierdo de la ecuación, y el símbolo al que se refiere el atributo pertenece a la cabeza de la producción (establecimiento del valor del atributo).
- El atributo aparece en el lado derecho de la ecuación, y el símbolo al que se refiere el atributo pertenece al cuerpo de la producción (consulta del valor del atributo).

Por su parte, un atributo será heredado cuando:

- El atributo aparece en el lado izquierdo de la ecuación, y el símbolo al que se refiere el atributo pertenece al cuerpo de la producción (establecimiento del valor del atributo).
- El atributo aparece en el lado derecho de la ecuación, y el símbolo al que se refiere el atributo pertenece a la cabeza de la producción (consulta del valor del atributo).

Esto permite asociar, a nivel local de cada ecuación, un conjunto de atributos sintetizados y un conjunto de atributos heredados con cada no terminal. Dicha asignación debe ser, entonces, consistente a lo largo de toda la especificación, en el sentido de que un mismo atributo no resulte, al mismo tiempo, sintetizado y heredado. Como resultado, será posible inferir el conjunto de atributos sintetizados y heredados de cada no terminal. Así mismo, en una vista, todos aquellos atributos asociados con símbolos no de núcleo se considerarán atributos correspondientes a dicha vista. Así mismo, todos los atributos de símbolos de núcleo no cualificados con nombre de vista se considerarán también atributos en dicha vista. Por último, los atributos cualificados se considerarán atributos en la vista correspondiente indicada en la cualificación. Como consecuencia, será posible determinar, además, para cada atributo, la vista a la que pertenece.

De esta forma, en una semántica gramatical, deben cumplirse las siguientes restricciones:

- Una semántica gramatical describe la semántica de una estructura gramatical, por lo que dicha estructura debe existir con las mismas reglas de producción en la semántica gramatical.
- Cada semántica gramatical sólo puede estar asociada a una única estructura gramatical, y cada estructura gramatical sólo puede tener una única semántica gramatical por vista asociada a dicha estructura. Si una estructura gramatical, para una de sus vistas asociadas, no posee semántica alguna, no es necesario asociar una semántica gramatical alguna, pues será vacía.

- En cada producción, cada atributo local sintetizado de la cabeza debe tener asignado una ecuación (y sólo una).
- En cada producción, cada atributo local heredado para cada ocurrencia de símbolo del cuerpo debe tener asignada una ecuación (y sólo una).
- Los símbolos terminales/no terminales/de núcleo referidos en las ecuaciones semánticas de una producción, junto sus números de ocurrencia, deben existir y ser correctos.
- En las partes derechas de las ecuaciones únicamente será posible referir a atributos sintetizados de las ocurrencias de los símbolos del cuerpo, así como a atributos heredados de la cabeza. Dichos atributos podrán ser tanto atributos locales como atributos externos (introducidos por otras vistas).
- Los terminales únicamente tienen atributos léxicos, los cuales tienen la misma condición que los sintetizados. En XLOP3, el terminal XML ***#pcdata*** posee sólo un atributo por defecto denominado *text*. Dicho atributo aloja el contenido textual del documento XML designado por *#pcdata*. Las etiquetas de cierre no poseen atributos. En las etiquetas de apertura, sus nombres de atributos se corresponden con los nombres de atributos respectivos definidos en el documento XML para dicha etiqueta (en XLOP3 no se realiza comprobación al respecto).

Es interesante remarcar que los atributos que aparecen sin cualificar deben ser locales a la semántica gramatical, y, por tanto, a la vista correspondiente. Esto quiere decir que todos estos atributos automáticamente poseerán la referencia de vista, que se expresa explícitamente añadiendo el nombre de vista como sufijo entre corchetes, y estarán asociados a dicha vista particular. En otras palabras, en dos vistas con la misma estructura gramatical, un atributo puede aparecer con el mismo nombre y referido al mismo terminal/no terminal, pero serán dos atributos diferentes. La referencia de vista, permite desambiguar unívocamente entre atributos correspondientes a otras vistas, aunque esto es únicamente aplicable a los atributos de los símbolos de núcleo (en otras palabras, los atributos de los símbolos no de núcleo son siempre locales a la vista correspondiente).

La Figura 5.5.3 presenta las especificaciones XLOP3 para las semánticas gramaticales de las fórmulas aritméticas, que añaden semántica a las estructuras gramaticales de la Figura 5.5.2, de acuerdo con la gramática multivista elaborada en la sección 5.2.4.

Vista env	Vista idr	Vista eva
Semantics for f of env { **f ::= <f> exp </f> {**eh of exp = mkenv(ids[idr] of exp);**}** **}**		**Semantics for f of eva {** **f ::= <f> exp </f> {** v of f = printResult(v of exp;)**}** **}**

Semantics for exp of env {
exp ::= opnd rexp {
eh of opnd = eh of exp;
eh of rexp = eh of exp;**}**
rexp ::= op opnd rexp {
eh of opnd = eh of rexp(0);
eh of rexp(1) = eh of rexp(0);**}**
rexp ::= {}
op ::= a {}
op ::= m {}
}
	Semantics for exp of idr {
	exp ::= opnd rexp {
	ids of exp = union(ids of opnd,
	ids of rexp);**}**
	rexp ::= op opnd rexp {
	ids of rexp(0) = union(ids of opnd,
	ids of rexp(1));**}**
	rexp ::= {ids of rexp = empty();**}**
	op ::= a {}
	op ::= m {}
	}
		Semantics for exp of eva {
		exp ::= expr {v of exp = v of expr;**}**
		expr ::= expr a term {
		v of expr(0) = add(v of expr(1),
		v of term);**}**
		expr ::= term{v of expr = v of term;**}**
		term ::= term m opnd {
		v of term(0) = mul(v of term(1),
		v of opnd);**}**
		term ::= opnd {
		v of term = v of opnd;**}**
		}

Semantics for opnd of env { **opnd ::= n {}** **opnd ::= v {}** **opnd ::= p {** eh of p = eh of opnd;**}** **}**	**Semantics for opnd of idr {** **opnd ::= n {**ids of opnd = empty();**}** **opnd ::= v {** ids of opnd = single(id of v);**}** **opnd ::= p {**ids of opnd = ids of p;**}** **}**	**Semantics for opnd of eva {** **opnd ::= n {**v of opnd = val(v of n);**}** **opnd ::= v {**v of opnd = valueOf(id[idr] of v, eh[env] of opnd);**}** **opnd ::= p {**v of opnd = v of p;**}** **}**
		Semantics for n of eva { **n ::= <Num> #pcdata </Num> {** v of n = text of #pcdata;**}** **}**
	Semantics for v of idr { **v ::= <Var> #pcdata </Var> {** id of v = text of #pcdata;**}** **}**	
Semantics for p of env { **p ::= exp {** eh of exp = eh of p;**}** **}**	**Semantics for p of idr {** **p ::= exp {** ids of p = ids of exp;**}** **}**	**Semantics for p of eva {** **p ::= exp {** v of p = v of exp;**}** **}**

Figura 5.5.3. Semánticas gramaticales en XLOP3 de las vistas env, idr, y eva.

Las funciones semánticas que figuran en las semánticas gramaticales permiten interconectar las tareas de procesamiento con el marco de aplicación del usuario. En este caso, las funciones que aparecen en la Figura 5.5.3 se implementarán en la clase semántica aportada por el usuario, implementando las siguientes funciones:

- *mkenv(in: conjunto)*: tabla. Crea y devuelve la tabla de variables a partir del conjunto de nombres de variables (in) y realiza las peticiones al usuario para que establezca el valor de dichas variables.

- *empty():* conjunto. Crea y devuelve un conjunto vacío.

- *single(in: cadena)*: conjunto. Crea y devuelve un conjunto con la cadena de entrada como único elemento.
- *union(a: conjunto, b: conjunto):* conjunto. Realiza la operación unión entre conjuntos y devuelve el conjunto resultado de la operación.
- *add(a: número, b: número)*: número. Realiza la suma de dos números y devuelve el resultado numérico.
- *mul(a: número, b: número)*: número. Realiza la multiplicación de dos números y devuelve el resultado numérico.
- *val(in: cadena)*: número. Convierte una cadena en un valor numérico y devuelve dicho valor.
- *valueOf(id: cadena, t: tabla)*: número. Busca en la tabla (t) la clave dada por la cadena (in) y devuelve su valor numérico asociado.
- *printResult(n: número)*: número. Imprime por pantalla el número que toma como parámetro y lo devuelve.

5.5.3 Uso y funcionamiento

En esta sección se describe el uso y el funcionamiento del entorno XLOP3 (se toma como ejemplo el caso de estudio de las fórmulas aritméticas).

XLOP3 toma como entrada una descripción de una gramática multivista, expresada en términos de su lenguaje de especificación, posiblemente distribuida en múltiples unidades físicas de almacenamiento (archivos), y constituida por:

- La gramática EBNF de base en la que basar la comprobación de la conformidad (opcional).
- Múltiples estructuras gramaticales que configuran las correspondientes estructuras sintácticas de las vistas.
- Múltiples semánticas gramaticales asociadas con dichas estructuras.

Entonces, opera en base a los siguientes pasos:

- *Tratamiento de las estructuras gramaticales.* Este paso realiza un análisis sintáctico de las especificaciones XLOP3 de estructuras gramaticales de la gramática de atributos multivista. Durante este proceso se construye un *modelo estructural* sobre el que operar posteriormente.

- *Comprobación de la conformidad y la no ambigüedad*. Este paso realiza la comprobación de la conformidad entre las estructuras gramaticales de la gramática de atributos multivista respecto la gramática EBNF aportada. En XLOP3 la gramática EBNF de base es, de hecho, opcional. En caso de no proporcionarse, XLOP3 permite utilizar la generalización del método de comprobación de conformidad descrita en el Capítulo 4 para comprobar la conformidad de las gramáticas asociadas a las distintas vistas. Así mismo, XLOP3 comprueba también la no ambigüedad de las gramáticas, mediante la construcción de los correspondientes autómatas LALR(1) y la comprobación de que dichos autómatas no contienen conflictos.

- *Tratamiento de las semánticas.* Este paso realiza un análisis sintáctico de las especificaciones XLOP3 de semánticas gramaticales. Durante este proceso se construye un *modelo semántico* sobre el que operar posteriormente. En este paso también comprueba las diferentes restricciones sobre la semántica (correcta referencia de atributos, correcto uso de las ecuaciones, etc.).

- *Generación de la aplicación.* Este paso genera y encapsula los elementos necesarios que constituirán la aplicación de procesamiento XML final. Estos elementos constan de la clase semántica, la lógica específica de la aplicación, y el entorno de ejecución y depuración XLOP3. Durante este paso, se verifica que la clase semántica realiza correctamente la interconexión entre la capa lingüística y la capa de la lógica de la aplicación. Así mismo, se configura también el entorno de ejecución y depuración que formará parte de la aplicación de procesamiento XML generada.

Estos cuatro pasos consecutivos son realizados por XLOP3 de manera automática, requiriendo para ello una configuración del entorno correcta, en función a la aplicación de procesamiento XML que se desea generar.

5.5.3.1 Configuración del entorno

El entorno XLOP3 está implementado en Java, y las aplicaciones desarrolladas bajo el entorno deben soportar la misma plataforma. Para que el entorno pueda generar una aplicación de procesamiento XML en los cuatro pasos descritos anteriormente, éste debe configurarse de manera correcta y específica según la aplicación concreta considerada. Para el cumplimiento de tal finalidad, el entorno dispone de una interfaz gráfica de usuario (Figura 5.5.4), en donde, por una parte, se configura la capa lingüística de la aplicación, y por otra parte se configura la capa de la lógica de la aplicación.

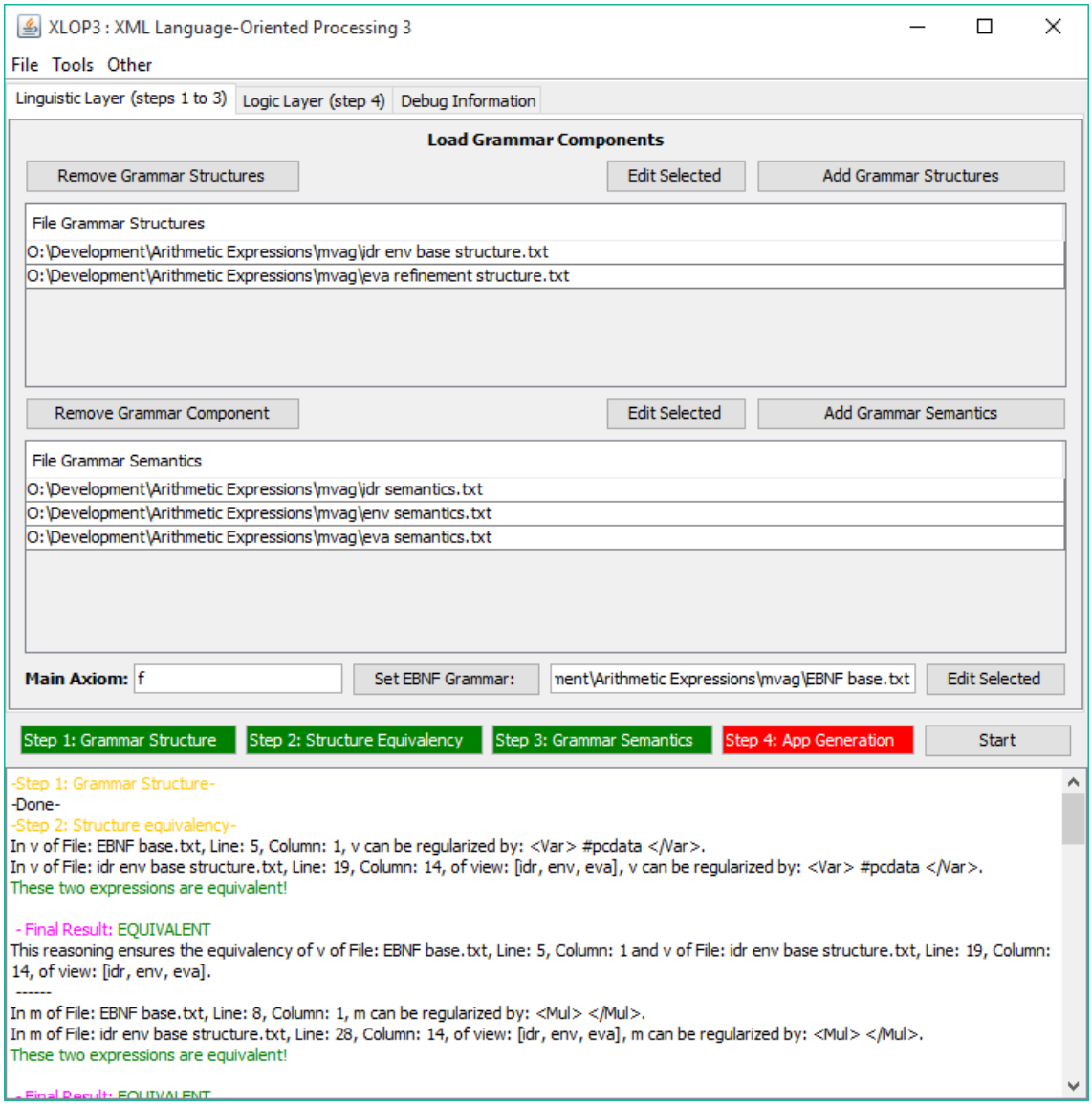


Figura 5.5.4. Interfaz del entorno XLOP3: configuración de la capa lingüística de la aplicación.

La capa lingüística se configura a través de la pestaña *Linguistic Layer (steps 1 to 3)*, en donde debe aportarse la ubicación de las especificaciones gramaticales multivista o, simplemente, especificaciones multivista. En la Figura 5.5.4 se muestra dicha configuración para la aplicación de las fórmulas aritméticas, donde pueden observarse más de un documento de especificación. Las especificaciones multivista XLOP3 son documentos textuales en donde los desarrolladores describen una gramática de atributos multivista mediante la gramática EBNF de base, las estructuras gramaticales, y las semánticas gramaticales asociadas a dichas estructuras. Como ya se ha indicado anteriormente, XLOP3 permite distribuir dichas unidades lógicas en diferentes archivos, aunque cada archivo debe contener únicamente un tipo de unidad (es decir, no pueden mezclarse, por ejemplo, semánticas gramaticales con estructuras gramaticales).

Para el ejemplo de las fórmulas aritméticas, las estructuras gramaticales de las fórmulas aritméticas, cuyas ubicaciones deben constar en la tabla *File Grammar Structures* del entorno, se han dividido en dos archivos:

- El archivo "idr env base structure.txt", que contiene todas las estructuras gramaticales de las vistas idr y env, de la Figura 5.5.2, con la misma sintaxis. Se ha denominado *estructura base*, porque define la estructura sintáctica básica del lenguaje.

- El archivo "eva refinement structure.txt", que contiene exclusivamente la estructura gramatical de la vista eva, de la Figura 5.5.2, definida bajo el símbolo **exp**. Se ha denominado *estructura refinada*, porque define una estructura más elaborada para un procesamiento más específico que la básica, en la que se abordan los aspectos de prioridad y asociatividad de operadores.

La comprobación de la conformidad entre estructuras gramaticales puede realizarse manualmente utilizando la herramienta *Grammar Equivalence Checker* introducida en el Capítulo 4, a la que puede accederse a través de la pestaña *Tools – Grammar Equivalence Checker*. No obstante, para la comprobación automática respecto a una gramática EBNF, la ubicación de su archivo de especificación debe configurarse con el botón *Set EBNF Grammar*.

Respecto a las semánticas gramaticales de las fórmulas aritméticas, sus ubicaciones deben constar en la tabla *File Grammar Structures* del entorno. Para el ejemplo, éstas se han dividido en tres archivos correspondientes a cada una de las tres vistas.

- El archivo "idr semantics.txt", que contiene las semánticas gramaticales de la vista **idr** de la Figura 5.5.3 con la misma sintaxis.

- El archivo "env semantics.txt", que contiene las semánticas gramaticales de la vista **env** de la Figura 5.5.3 con la misma sintaxis.

- El archivo "eva semantics.txt", que contiene las semánticas gramaticales de la vista **eva** de la Figura 5.5.3 con la misma sintaxis.

El axioma de la gramática multivista debe especificarse en el campo de texto *Main Axiom*. Este axioma es el símbolo **f** para las fórmulas aritméticas. Configurada la capa lingüística, el entorno XLOP3 será capaz de realizar los tres primeros pasos del proceso de generación, como se muestra en la Figura 5.5.4, iniciados con el botón *Start*. Como puede observarse, en verde figuran los pasos que se han realizado correctamente, y en el campo de texto inferior se emite información sobre el proceso realizado y los resultados obtenidos, así como sobre cualquier posible error que haya sido detectado. Si se produce un error en alguno de los pasos que impida la continuación del proceso global, dichos pasos se señalarán en rojo. Un paso señalado en morado indicará que existen posibles condiciones en la especificación que podrían dar lugar a un comportamiento de ejecución de la aplicación generada diferente al esperado. No

obstante, en todo momento se mostrará información relevante para la depuración de las especificaciones (sección 5.5.3.2) en la pestaña *Debug Information*.

Para poder realizarse el cuarto paso debe configurarse también la capa de la lógica de la aplicación en la interfaz de XLOP3 (Figura 5.5.5). La capa lógica se configura a través de la pestaña *Logic Layer (step 4)*. En ella debe aportarse la ubicación de los siguientes elementos: clase semántica, bibliotecas del marco de la aplicación y, recursos de la aplicación.

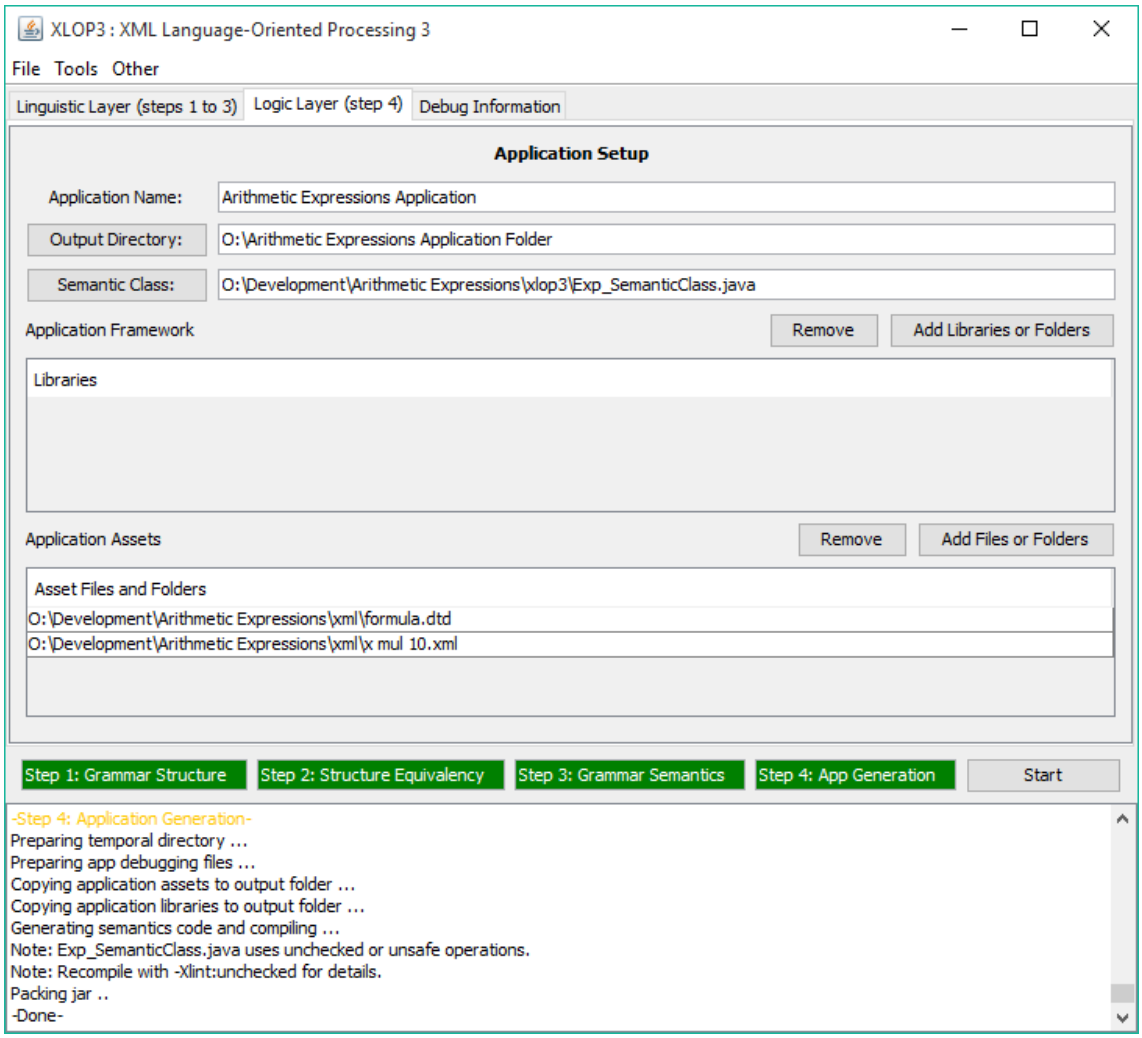


Figura 5.5.5. Interfaz del entorno XLOP3: configuración de la capa de la lógica de la aplicación.

En concreto, en la Figura 5.5.5 se muestra la configuración para la aplicación de las fórmulas aritméticas, que no posee un marco de aplicación propio pues, por su simplicidad, toda la lógica operacional queda codificada en la clase semántica. El nombre que poseerá la aplicación generada se introduce en el campo *Application Name*. La carpeta que incluirá las bibliotecas y recursos de la aplicación generada se establece en el campo *Output Directory*. La ruta de la clase semántica se define a través del campo *Semantic Class*. Por último, las ubicaciones de las bibliotecas del marco de la aplicación se introducen en la tabla *Application*

Framework, y las ubicaciones de los recursos adicionales de los que hará o podrá hacer uso la aplicación (DTD, instancias XML, imágenes, audio, etc.), se introducen en la tabla *Application Assets*. Como puede observarse en la Figura 5.5.5, ahora el cuarto paso de generación de la aplicación podrá realizarse correctamente, figurando su estado en verde.

Arithmetic Expressions Application Folder\temp\SemanticClassInterface.java

```
package xlop3;
public interface SemanticClassInterface {
  public Object union(Object a0, Object a1);
  public Object empty();
  public Object val(Object a0);
  public Object single(Object a0);
  public Object valueOf(Object a0, Object a1);
  public Object add(Object a0, Object a1);
  public Object mul(Object a0, Object a1);
  public Object printResult(Object a0);
  public Object mkenv(Object a0);
}
```

Exp_SemanticClass.java

```
package xlop3;
import java.util.Map;  import java.util.HashMap;  import java.util.HashSet;  import java.util.Set;
public class Exp_SemanticClass implements SemanticClassInterface {
  public Object mkenv(Object set) {
    HashMap<String,Integer> table = new HashMap<>();
    for (String var:(Set<String>)set) { System.out.println("Enter value for variable:"+var);
      Integer value = Integer.valueOf(System.console().readLine()); System.console().flush();  table.put(var,value);
    }
    return table;
  }
  public Object empty() {
    return new HashSet();
  }
  public Object single(Object element) {
    HashSet set = new HashSet();
    set.add(element);
    return set;
  }
  public Object union(Object set_a, Object set_b) {
    ((Set)set_a).addAll(((Set)set_b));
    return set_a;
  }
  public Object add(Object op_a, Object op_b) {
    return (int)op_a +(int)op_b;
  }
  public Object mul(Object op_a, Object op_b) {
    return (int)op_a * (int)op_b;
  }
  public Object val(Object value) {
    return Integer.valueOf((String)value);
  }
  public Object valueOf(Object key, Object table) {
    return ((Map)table).get(key);
  }
  public Object printResult(Object n) {
    System.out.println("The result is:"+n);
    return n;
  }
}
```

Figura 5.5.6. Clase semántica de las fórmulas aritméticas e interfaz semántica que implementa generada por XLOP3.

En el proceso que se realiza durante el cuarto paso, interviene la compilación Java de la clase semántica con las bibliotecas del marco de la aplicación y el código generado para configurar el procesador del entorno de ejecución. Este código generado por XLOP3 es en lo que se transforma las semánticas gramaticales de la gramática de atributos multivista: la factoría de nodos específica para la construcción de bosques de análisis atribuidos, a la que se ha hecho referencia en la sección 5.3. Respecto a la clase semántica, XLOP3 produce una interfaz para dicha clase y que debe implementar (ubicándola en la carpeta *temp* de la carpeta establecida en *Output Directory*), para así garantizar que las funciones semánticas definidas en la gramática de atributos multivista poseen métodos que las implementan. Por cuestiones de diseño, ambas clases deben pertenecer al paquete *xlop3*, y el sistema donde se ejecuta el entorno debe poseer el kit de desarrollo Java SE 8 o superior, señalando en la pestaña *Other – Set Java JDK* dónde se ubica dicho kit de desarrollo. Por último, toda la configuración realizada en el entorno puede ser guardada y cargada, respectivamente, a través de la pestaña *File – Open Configuration/Save Configuration.*

Para las fórmulas aritméticas, la interfaz semántica producida es la que se muestra en la Figura 5.5.6, figura que también contempla la clase semántica que implementa la funcionalidad detallada en la sección 5.5.2.3.

5.5.3.2 Depuración de especificaciones multivista

XLOP3 integra un motor para realizar todas las comprobaciones sintácticas y de buena formación de cada una de las especificaciones estructurales y semánticas aportadas por el usuario. Los informes de errores que emite el entorno son específicos a cada archivo de especificación, y se muestran en la zona inferior de su interfaz en color morado.

Para facilitar la depuración de las especificaciones, el entorno aporta un pequeño editor denominado *Quick Grammar Editor*. Este editor permite modificar el contenido de los archivos de especificación, así como realzar fragmentos textuales en morado para indicar con exactitud en qué puntos de la especificación se han encontrado los errores o inconsistencias. La Figura 5.5.7 muestra la interfaz y un ejemplo de funcionamiento del editor, en donde puede observarse cómo se muestran y señalan los errores para un caso de especificación semántica mal formada. En la parte inferior de la figura puede observarse que el entorno XLOP3 encuentra varios errores en más de un archivo de especificación, mostrando, para cada uno, la descripción del error, la fila, la columna y el archivo en el cual se produjo. Para abrir uno de estos archivos con el editor de manera directa, deberá señalarse el archivo deseado en la tabla *File Grammar* Structures o *File Grammar Semantics* y pulsar el botón *Edit Selected*. Otra manera de abrir una instancia adicional del editor es mediante la pestaña *Tools – Quick Grammar Editor*.

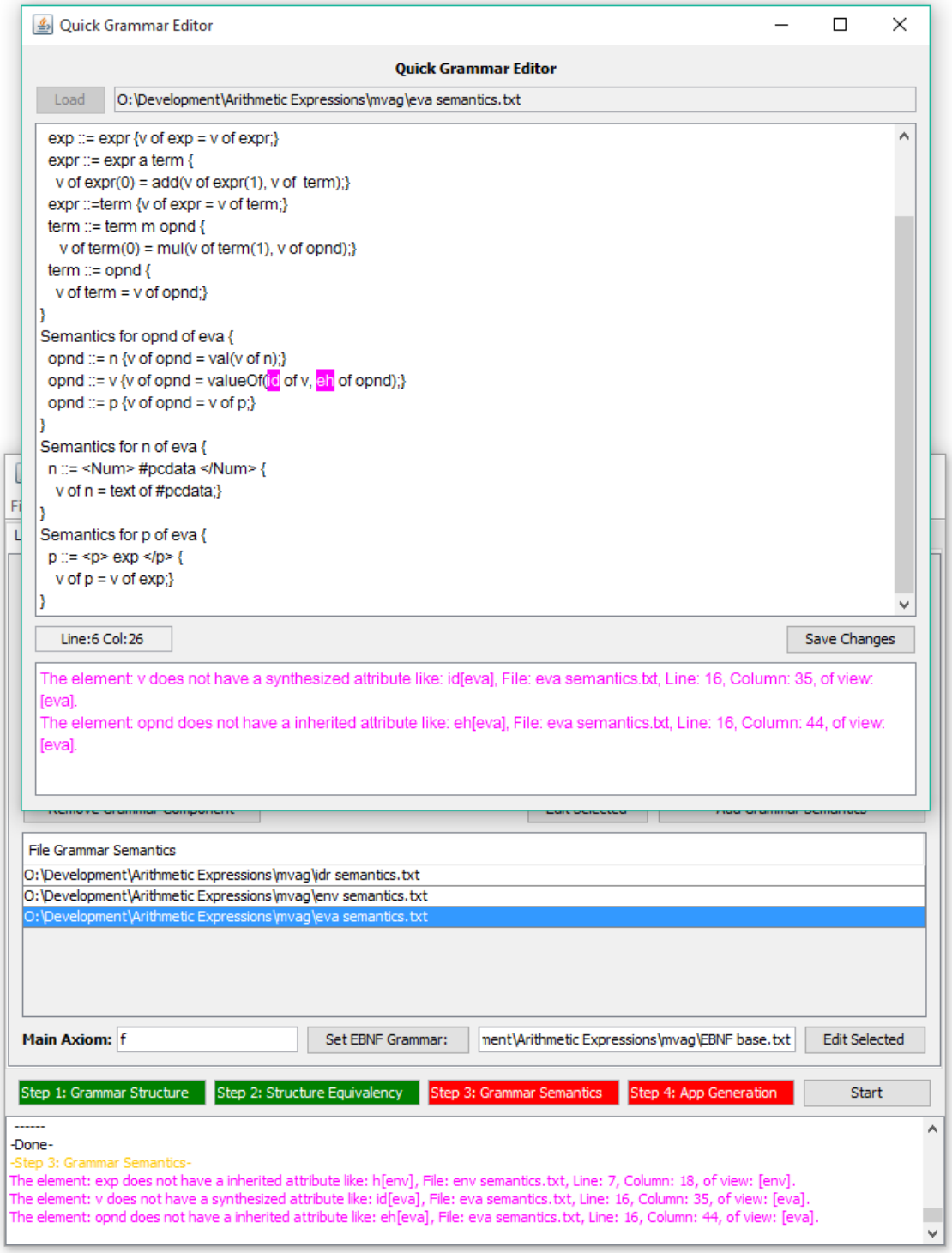


Figura 5.5.7. Editor de archivos de especificación XLOP3: Quick Grammar Editor (ventana superior) del entorno XLOP3 (parte de ventana inferior).

XLOP3 maneja las especificaciones internamente de manera distinta a como fueron descritas. Para generar la gramática de atributos multivista descrita mediante una especificación multivista se requiere previamente realizar la desambiguación de sus términos. Esto se traduce en la necesidad de realizar una desambiguación de nombres de símbolos no terminales y nombres de atributos semánticos de los símbolos de la especificación multivista, a fin de que la gramática de atributos multivista resultante de unir todos los fragmentos gramaticales de todas las vistas descritas no mezcle identificadores cuyo significado tiene sentido sólo a un nivel local (de fragmento gramatical o de vista). XLOP3 simplifica ésta tarea, realizándola automáticamente para el usuario, de la siguiente manera:

- La desambiguación de los nombres de atributos semánticos se realiza añadiendo a todos ellos un sufijo con el nombre de vista bajo el que son definidos. Estos nombres de atributos tienen sentido únicamente a nivel local de vista. Este sufijo contiene el nombre de vista de la semántica gramatical bajo la cual se describe tal atributo. La desambiguación automática ocurre únicamente cuando este sufijo no es aportado explícitamente por el usuario. El formato empleado por XLOP3 es el siguiente:
    - –nombre atributo semántico-**[**-nombre de vista-**]**.
- La desambiguación de los nombres de símbolos no terminales se realiza añadiendo a todos ellos un prefijo con un formato especial. Como ya se ha indicado, los nombres de estos símbolos tienen sentido únicamente a nivel local de estructura gramatical. No obstante, los símbolos terminales y los símbolos de núcleo son únicos a nivel global. Este prefijo contiene las iniciales de vistas ordenadas correspondientes a la primera letra de los nombres de vista de la estructura gramatical bajo la cual tal símbolo es descrito, seguido de un identificador numérico unívocamente otorgado a cada estructura gramatical existente. Esta desambiguación automática ocurre siempre. El formato empleado por XLOP3 es el siguiente:
    - **[**-iniciales de vistas-*****-identificador numérico de estructura gramatical-**]**

No obstante, el renombrado realizado por XLOP3 afecta directamente al proceso de depuración de especificaciones. Como puede observarse, la desambiguación es efectiva a la hora de mostrar errores o inconsistencias por el entorno, como puede observarse en la Figura 5.5.7. Para conocer cómo se ha realizado la desambiguación, se deberá consultar la pestaña *Debug Information* del entorno. En ella se mostrará la siguiente información de depuración ya desambiguada:

- Todos los símbolos de núcleo de la gramática multivista (también referidos como axiomas de fragmentos).
- Cada una de las vistas sintácticas acondicionadas con su semántica de vista correspondiente.

- La gramática incontextual multivista completa, resultante de unificar todas las estructuras gramaticales.
- La tabla de atributos. En ella se muestran cuáles son los atributos sintetizados y heredados de cada símbolo de la gramática multivista.
- La gramática de atributos multivista completa, resultante de unificar todas las estructuras y semánticas gramaticales.

Para el caso de ejemplo de las fórmulas aritméticas, la gramática de atributos desambiguada por XLOP3 tomaría el aspecto de la Figura 5.5.8.

Gramática de atributos multivista desambiguada

```
f ::= <f> exp </f> {
  f.v[eva] = printResult(exp.v[eva])
  exp.eh[env] = mkenv(exp.ids[idr])
 }
 exp ::= opnd [ie*1]rexp {
  exp.ids[idr] = union(opnd.ids[idr],[ie*1]rexp.ids[idr])
  [ie*1]rexp.eh[env] = exp.eh[env]
  opnd.eh[env] = exp.eh[env]
 }
 exp ::= [e*2]expr {
  exp.v[eva] = [e*2]expr.v[eva]
 }
 [ie*1]rexp ::= [ie*1]op opnd [ie*1]rexp {
  [ie*1]rexp.ids[idr] =
union(opnd.ids[idr],[ie*1]rexp(1).ids[idr])
  [ie*1]rexp(1).eh[env] = [ie*1]rexp.eh[env]
  opnd.eh[env] = [ie*1]rexp.eh[env]
 }
 [ie*1]rexp ::= {
  [ie*1]rexp.ids[idr] = empty()
 }
 [ie*1]op ::= a
 [ie*1]op ::= m
 [e*2]expr ::= [e*2]expr a [e*2]term {
  [e*2]expr.v[eva] = add([e*2]expr(1).v[eva],[e*2]term.v[eva])
 }
 [e*2]expr ::= [e*2]term {
  [e*2]expr.v[eva] = [e*2]term.v[eva]
 }
 [e*2]term ::= [e*2]term m opnd {
  [e*2]term.v[eva] = mul([e*2]term(1).v[eva],opnd.v[eva])
 }
 [e*2]term ::= opnd {
  [e*2]term.v[eva] = opnd.v[eva]
 }
 opnd ::= n {
  opnd.ids[idr] = empty()
  opnd.v[eva] = val(n.v[eva])
 }
 opnd ::= v {
  opnd.ids[idr] = single(v.id[idr])
  opnd.v[eva] = valueOf(v.id[idr],opnd.eh[env])
 }
 opnd ::= p {
  opnd.ids[idr] = p.ids[idr]
  p.eh[env] = opnd.eh[env]
  opnd.v[eva] = p.v[eva]
 }
 n ::= <Num> #pcdata </Num> {
  n.v[eva] = #pcdata.text
 }
 v ::= <Var> #pcdata </Var> {
  v.id[idr] = #pcdata.text
 }
 p ::= <p> exp </p> {
  p.ids[idr] = exp.ids[idr]
  exp.eh[env] = p.eh[env]
  p.v[eva] = exp.v[eva]
 }
 a ::= <Sum> </Sum>
 m ::= <Mul> </Mul>
}
```

Figura 5.5.8. Gramática de atributos multivista desambiguada para las fórmulas aritméticas.

5.5.3.3 Generación del procesador XML de la aplicación

El entorno XLOP3 posee un generador de código que configura específicamente el procesador de documentos XML que poseerá cada aplicación de procesamiento XML generada bajo dicho entorno. Esta configuración se genera a partir de la especificación multivista de la aplicación, resultando en varios componentes directamente accesibles por el entorno de ejecución y depuración XLOP3, y componentes en código Java que deben ser compilados para su uso directo, formando bibliotecas para dicho entorno de ejecución y depuración. La Figura 5.5.9 muestra este proceso de generación, en donde finalmente la aplicación de procesamiento XML quedará lista para funcionar con documentos de entrada XML.

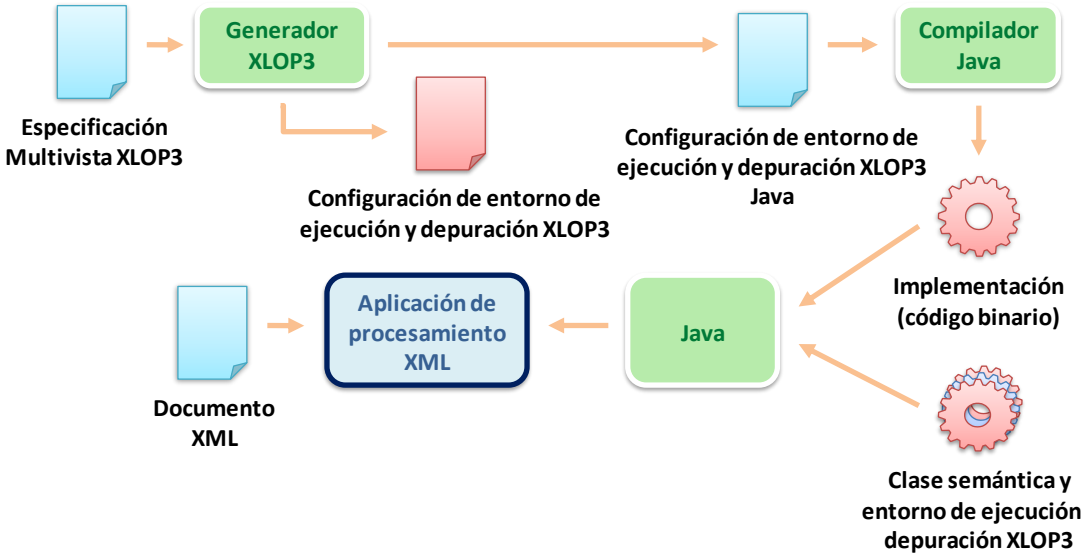


Figura 5.5.9. Proceso de generación de XLOP3 de los componentes del entorno de ejecución y depuración para las aplicaciones de procesamiento XML.

Para que el proceso de generación se lleve a cabo correctamente, el entorno XLOP3 debe haberse configurado correctamente de la manera que ya se expuso en la sección 5.5.3.1. Si la especificación multivista es correcta y la clase semántica aportada realiza correctamente la interconexión entre la capa lingüística y la capa de la lógica de la aplicación, entonces el cuarto paso de generación de la aplicación producirá los siguientes componentes que constituirán la aplicación de procesamiento XML (en la carpeta de salida de la aplicación indicada en el entorno):

- Bibliotecas del marco de la aplicación y recursos. Las mismas bibliotecas y recursos cuya ubicación se establece en la tabla *Application Framework* y en la tabla *Application Assets* del entorno XLOP3. En el caso de ejemplo de las fórmulas aritméticas, sus recursos son el documento de definición de lenguaje XML (DTD) e instancias XML, pero no posee marco de aplicación explícito debido a que toda la lógica de la aplicación reside en su clase semántica (Figura 5.5.6).

- Accesos directos para la ejecución de la aplicación. Funcionarán aportando una instancia XML. Permiten realizar la ejecución de la aplicación iniciando el entorno de ejecución directamente, o iniciando el entorno de depuración. En la sección 5.5.4 se muestran más detalles y opciones de lanzamiento.

- Carpeta temporal (*temp*). En ella figurará la interfaz generada por XLOP3 cuyos métodos deben ser implementados en la clase semántica de la aplicación: *SemanticClassInterface* (sección 5.4.1). En el caso de ejemplo de las fórmulas aritméticas esta interfaz sería como se muestra en la Figura 5.5.6.

- Carpeta de bibliotecas (*lib*). Contendrá distintos componentes que son parte del entorno de ejecución y depuración XLOP3, en donde ciertos elementos son específicos para la aplicación generada.

  En relación con la carpeta *lib*, el contenido específicamente incluido es el siguiente:

- Biblioteca *XLOP3.parsetable*. Biblioteca con funcionalidad para albergar y operar eficientemente sobre tablas de análisis sintáctico ascendente. Este componente permite interpretar las tablas de análisis sintáctico específicas de la aplicación.

- Tablas de análisis sintáctico. Tablas de análisis sintáctico generadas por XLOP3 a partir de las estructuras gramaticales de la especificación multivista. Estas tablas son específicas de la aplicación y necesarias para el funcionamiento de los algoritmos de análisis.

- Biblioteca *XLOP3.eval*. Biblioteca que aporta los modelos SPPFA para la construcción de bosques atribuidos, con funcionalidad para albergar y operar con atributos, valores y funciones semánticas.

- Biblioteca *XLOP3.sem_fac*. Contenedor de la clase semántica de la aplicación (compilada) y de la factoría de nodos SPPFA generada por XLOP3 a partir de las semánticas gramaticales de la especificación multivista (compilada). Esta biblioteca es específica de la aplicación.

- Biblioteca *XLOP3.runtime*. Biblioteca principal del entorno de ejecución y depuración que permite analizar el documento XML de entrada, realizar el proceso de construcción de los bosques de análisis sintáctico, atribuirlos, e iniciar el proceso de evaluación de los bosques atribuidos. Para ello, como núcleo posee el analizador MVGLR (aunque éste puede intercambiarse directamente por una implementación basada en el MVLR) con un componente constructor de bosques atribuidos, y un escáner XML basado en SAX. Así mismo, incluye el entorno de depuración para la aplicación, que se describirá en la sección 5.5.4.2.

- Biblioteca *XLOP3.utils* y recursos de depuración, que proporcionan estructuras de datos y operaciones auxiliares genéricas, una aplicación de visualización de grafos (yComp), y archivos con reglas gramaticales en formato textual y grafo de autómatas en formato VCG [Lemke 1993] específicos de la aplicación.

La biblioteca principal de ejecución *XLOP3.runtime* dispone de opciones de lanzamiento del procesador de documentos XML de la aplicación de manera directa o en modo de depuración. Si la aplicación generada debe ser iniciada directamente a través del procesador de documentos XML, se podrá realizar la ejecución de la aplicación simplemente a través de los accesos directos disponibles en la carpeta de generación. Estos accesos configuran el modo de operación del entorno de ejecución y depuración XLOP3 de la aplicación generada. En la siguiente sección se muestra con más detalle el funcionamiento del entorno de ejecución y depuración.

5.5.4 Entorno de ejecución y depuración

El entorno de ejecución es el componente principal de procesamiento XML para las aplicaciones generadas con XOP3, aportado y configurado por el entorno específicamente para cada una de estas aplicaciones. Permite un modo de ejecución directa y un modo de depuración. Ambos modos operan para iniciar el procesador de documentos XML de una manera determinada, y pueden ser configurados manualmente a través de los métodos de la biblioteca *XLOP3.runtime*. Si se desea iniciar el modo de ejecución directa, el método *main* de esta biblioteca deberá tener como argumento de entrada la ubicación del documento XML a procesar. En cambio, si se desea iniciar el modo de depuración, se deberá incluir el comando **-d** como argumento de entrada adicional a la declaración de la ruta del documento XML. No obstante, si se desea iniciar el procesador XML en otro instante concreto, desde otro proceso, bajo otras configuraciones, o conseguir otro modo de ejecución deseado, deberán realizarse invocaciones de los métodos de la biblioteca *XLOP3.runtime* desde el marco de la aplicación.

5.5.4.1 Ejecución directa de la aplicación

La ejecución directa de la aplicación inicia el procesador de documentos XML, en un primer paso orquestado por el analizador MVGLR y el componente constructor de bosques SPPFA, y en un segundo paso orquestado por la dinámica de evaluación de atributos.

La Figura 5.5.10 muestra un ejemplo de ejecución de la aplicación de las fórmulas aritméticas generada por XLOP3, utilizando una instancia XML que aloja la expresión: “x*10”. La información presentada por la aplicación se encuentra embebida entre la etiqueta -*-*Started*-*- y la etiqueta -*-*Finished*-*-. Como puede observarse, el proceso que se ha realizado internamente ha sido el deseado y descrito en la especificación gramatical multivista. Para ello, se capturaron los nombres de todas las variables (en este caso **x**), y se pidió al usuario el establecimiento de su valor (en este caso **9**). Posteriormente, con dicha información, se

sustituyó en la expresión "x*10" la variable **x** por el valor **9** y, finalmente, se realizó la multiplicación obteniendo como resultado: 90.

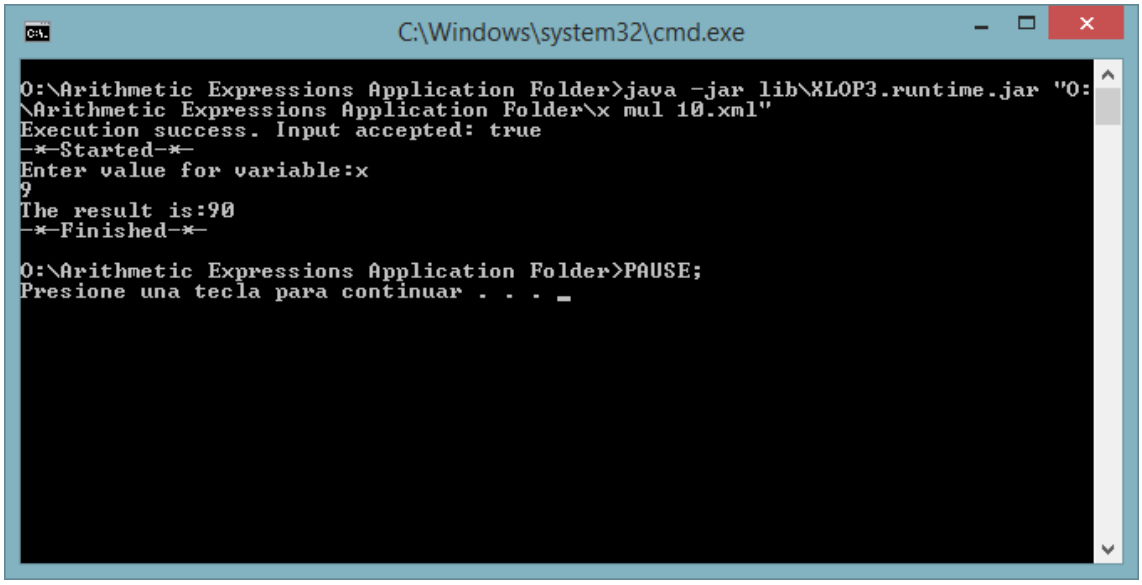


Figura 5.5.10. Ejemplo de ejecución de la aplicación de las fórmulas aritméticas generada con XLOP3, para el documento XML expresión "x*10".

5.5.4.2 Depuración de la aplicación

El entorno de depuración permite la ejecución paso a paso del procesador de documentos XML. Esta ejecución paso a paso está guiada por el analizador MVGLR en función de la tabla de análisis sintáctico obtenida a partir del multiautómata LALR(1) y el token de entrada aportado por el escáner XML SAX. La interfaz del entorno de depuración se muestra en la Figura 5.5.11, y permite procesar un documento XML con el botón *Open Input (XML)* paso por paso, por número salteado de pasos, o de manera completa, mostrando en cada momento el estado del proceso que se lleva a cabo.

El modo *paso por paso* se inicia con el botón *Start Step Parsing (GLR)*. Tras su activación, el estado del botón cambiará a *Ready to Parse* (posteriormente a *Next Step*), se iniciará el analizador léxico, y éste mostrará el siguiente token disponible para ser consumido por el analizador MVGLR. Cada pulsación del botón realizará un paso de ejecución, implicando:

- Consumir el siguiente token disponible, aportado por el analizador léxico y mostrado en el campo *Next Token*.
- Mostrar el token actual consumido, añadiéndolo a la pestaña *Parsed Input*, que mostrará todos los tokens consumidos.
- Señalar la acción o acciones realizadas en la tabla de análisis por el token consumido, en la pestaña *Parse Table*. Los números de estado de la tabla, de las

acciones de desplazamiento **Sh** y, de las celdas de no terminales, hacen referencia a los estados del multiautómata que pueden visualizarse en la pestaña *LALR(1) Automaton*. Los números de regla de las acciones de reducción **Re** hacen referencia a las producciones numeradas en la pestaña *Grammar Rules*.

- Detallar las acciones realizadas por el analizador sobre la pila grafo GSS, en el campo *Information*.
- Permitir visualizar el estado de la pila grafo GSS mediante el botón *View GSS*.
- Permitir visualizar el estado de construcción del bosque de análisis sintáctico SPPF mediante el botón *View SPPF*.

El modo salteado de pasos opera idénticamente al caso anterior, realizando el número de pasos de ejecución consecutivos deseado, reiniciando el estado del procesador y analizando el documento XML desde el inicio. Esta ejecución se realiza con el botón *Jump to Step* indicado en el campo a su derecha el número de pasos a realizar automáticamente.

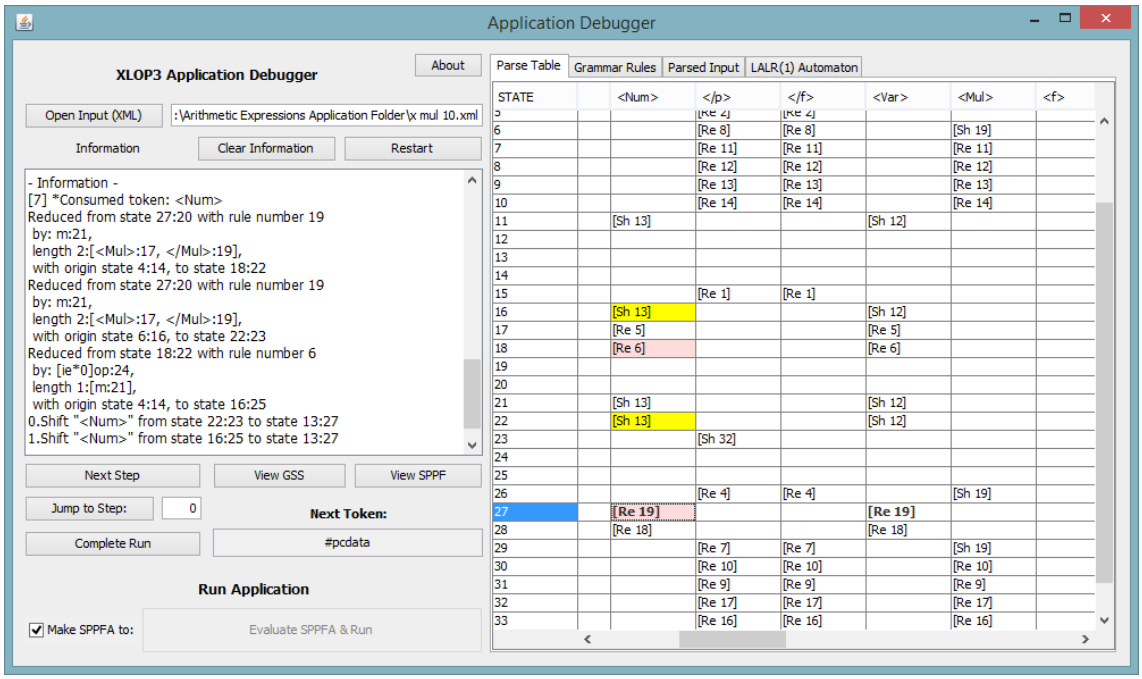


Figura 5.5.11. Entorno de depuración XLOP3.

El modo completo se realiza, por último, con el botón *Complete Run*. La ejecución se realiza completa, reiniciando el estado del procesador y analizando el documento XML desde el inicio hasta el final.

Los errores producidos durante el procesamiento de un documento XML se mostrarán en el campo *Information*. Mediante el botón *Restart* se reiniciará el analizador. Durante cualquier modo de ejecución podrá visualizarse el estado de la pila grafo GSS y la construcción del bosque SPPF mediante una herramienta visualizadora de grafos que interprete el formato VCG

[Lemke 1993]. Estos archivos son producidos automáticamente y quedan disponibles en la carpeta temporal (*temp*) de la aplicación de procesamiento XML. Los botones *View GSS* y *View SPPF* automatizan la visualización de estos grafos GSS y SPPF, respectivamente, mediante la herramienta [yComp]. La Figura 5.5.12 muestra dicha herramienta, donde puede apreciarse cómo es el estado de construcción del bosque SPPF del ejemplo de las fórmulas aritméticas a los diez pasos de ejecución del procesamiento del documento XML que alberga la expresión "x*10" (Figura 5.5.11).

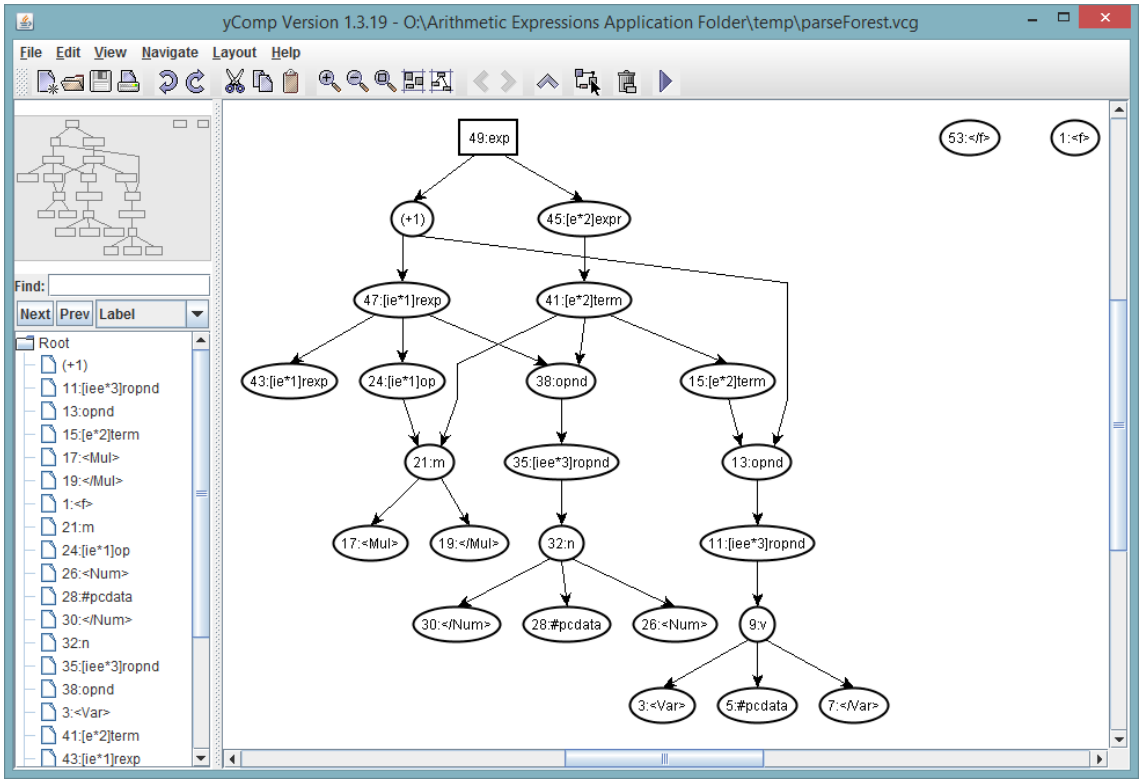


Figura 5.5.12. Visualizador de grafos yComp.

La sección inferior de la interfaz del depurador permite la ejecución de la aplicación. Para ello, es necesario realizar la construcción del bosque SPPFA en su totalidad, marcando la casilla *Make SPPFA* y ejecutando el procesamiento del documento XML de manera completa, mediante alguno de los tres modos de ejecución anteriormente expuestos. La ejecución de la aplicación (y/o continuación de la ejecución de la aplicación), se realiza con el botón *Evaluate SPPFA & Run*. El efecto que tendrá lugar será el iniciar el proceso de evaluación del bosque SPPFA, que consiste en solicitar el valor de los atributos sintetizados del nodo raíz del bosque, lo cual desencadenará un proceso de evaluación por demanda en el bosque atribuido. Durante este proceso, si las operaciones que se realizan o la aplicación que se ejecuta muestra o requiere información, ésta se visualizará y establecerá a través de ella o de la entrada y salida estándar, como en el caso de ejemplo de las fórmulas aritméticas (Figura 5.5.10).

5.5.5 Arquitectura

La construcción del entorno XLOP3 parte de lo aprendido mediante el desarrollo de su predecesor, XLOP 1.0. Ambos entornos tienen un objetivo similar: generar aplicaciones de procesamiento XML con gramáticas de atributos, marcando la separación entre dos capas bien diferenciadas para dichas aplicaciones, una capa lógica, y una capa lingüística, que se interconectan entre sí a través de una clase semántica. Para llevar a cabo dicho cometido:

- En la primera versión de XLOP, para gramáticas de atributos convencionales, existen diversas herramientas externas que pueden utilizarse. Es por ello que la arquitectura de XLOP 1.0 es bastante sencilla, al utilizar herramientas como JavaCC o CUP para la generación de traductores del mismo entorno y para las aplicaciones generadas.

- No obstante, XLOP3 introduce un enfoque metalingüístico nuevo: gramáticas de atributos multivista. Este enfoque implica realizar un rediseño casi total de la arquitectura original de XLOP. Así mismo, el carácter original de la propuesta de las gramáticas de atributos multivista imposibilita encontrar alguna herramienta externa que facilite la generación de los traductores de las aplicaciones generadas con gramáticas de atributos multivista.

En esta sección se comienza, por tanto, analizando la antigua arquitectura de XLOP 1.0 (sección 5.5.5.1), a fin de que sirva de base y motive la arquitectura más compleja de XLOP3. Seguidamente se muestra dicha nueva arquitectura (sección 5.5.5.2). A lo largo de esta sección, se utilizarán de forma extensiva referencias a otras partes de la memoria a fin de evidenciar cómo XLOP3 realiza de manera efectiva las propuestas descritas en esta tesis.

5.5.5.1 Antigua arquitectura

La complejidad de la arquitectura de la primera versión de XLOP 1.0 reside en el proceso de generación que lleva a cabo. Tal y como se indicó en el Capítulo 3, este proceso consiste en generar, a partir de una especificación XLOP de gramática de atributos, una especificación CUP con esquemas de traducción para el generador de traductores CUP, en la cual se codifica un mecanismo de evaluación dirigido por los datos. Mediante una especificación CUP, el generador de CUP es capaz de construir todos los elementos necesarios (autómata LALR(1), tablas de análisis, etc.) para la correcta ejecución del traductor de tipo LR que produce. Mediante el uso de esta herramienta se simplifica enormemente la generación de los traductores de las aplicaciones de procesamiento XML generadas. Sin embargo, es interesante analizar el proceso interno que se realiza en XLOP para la generación, pues posee ciertos elementos que pueden ser de provecho para la construcción del entorno XLOP3.

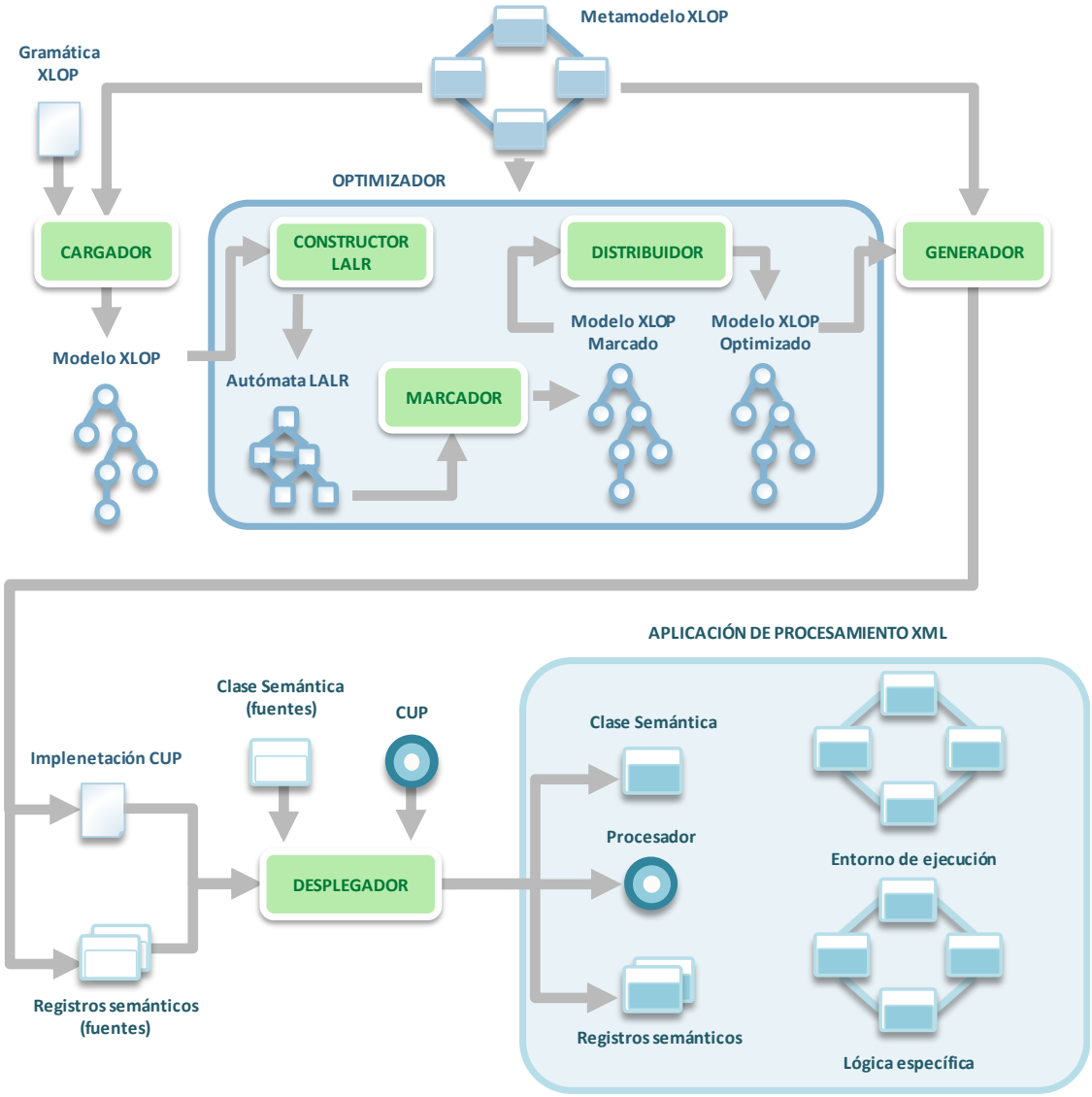


Figura 5.5.13. Arquitectura XLOP 1.0 con soporte de gramáticas de atributos.

La arquitectura de XLOP 1.0 con soporte de gramáticas de atributos convencionales se muestra en la Figura 5.5.13. El proceso interno que se realiza es el siguiente:

- Análisis y transformación de la gramática de atributos. La gramática de atributos se especifica mediante una notación propia de XLOP en un único archivo textual. Dicha gramática se analiza y transforma a un modelo gramatical. Este análisis y construcción del modelo se realiza mediante un traductor construido con la herramienta JavaCC. Durante este proceso, también se construye la tabla de atributos y se comprueban algunas de las restricciones contextuales para verificar que la gramática está bien formada. Otras de estas comprobaciones se realizan posteriormente sobre el modelo ya construido.

- Introducción de marcadores y construcción del autómata LALR(1). Los marcadores consisten en nuevos símbolos no terminales, definidos mediante producciones nulas, que se introducen en las reglas gramaticales, para alojar atributos semánticos cuyos valores pueden estar disponibles o ser calculados de manera adelantada [Aho et al. 2006]. Con ello se aumenta la eficiencia del proceso de evaluación dirigido por los datos del traductor generado para las aplicaciones mediante CUP. Para ello, se requiere realizar la construcción del autómata LALR(1) y calcular los grafos de dependencias entre elementos de los estados, cuyos análisis permiten decidir en qué puntos de la gramática es posible alojar marcadores [Temprado 2009]. Como resultado, el modelo de objetos XLOP se modifica con la inclusión de dichos marcadores, obteniendo un modelo intermedio marcado y refinándose hasta obtener finalmente un modelo optimizado.

- Generación del procesador de documentos XML de la aplicación. A partir del modelo de objetos XLOP optimizado se genera una especificación CUP, cuya herramienta genera, a partir de dicha especificación, la implementación del traductor, cuyos esquemas de traducción codifican las ecuaciones semánticas. Por otra parte, del modelo de objetos XLOP se generan los registros semánticos, necesarios para el traductor, que permiten alojar y acceder a los valores de los atributos semánticos. Este traductor compilado constituye el procesador, que es adaptado a XML mediante su conexión con un escáner XML basado en SAX, conexión realizada por el entorno de ejecución y que inicia el proceso de análisis. Las funciones definidas en el modelo de objetos XLOP se invocan mediante reflexión a través de la clase semántica aportada por el usuario, haciendo de esta clase una conexión con el marco de aplicación o lógica específica. Todos estos elementos conforman la aplicación de procesamiento XML generada.

Existe una versión posterior a XLOP 1.0, denominado XLOP 2.0. Esta versión fue fruto de una revisión consistente en el refinamiento de los algoritmos de introducción de marcadores y en el soporte de especificaciones modulares mediante fragmentos de gramáticas de atributos, siguiendo el enfoque descrito en [Sarasa & Sierra 2013-b]. Sin embargo, dicha versión no aporta una arquitectura nueva.

5.5.5.2 Nueva arquitectura

La nueva arquitectura de XLOP3 es substancialmente más compleja que la de su predecesor. Para poder dar un soporte completo a las gramáticas de atributos multivista, muy pocos elementos de la arquitectura antigua son directamente reutilizables para el nuevo entorno. Las gramáticas de atributos multivista introducen una mayor carga computacional a la hora de procesar las especificaciones gramaticales, transformarlas a modelos de objetos, y realizar sobre ellas las comprobaciones de buena formación. Por el carácter modular de estas especificaciones, en esta ocasión no existe un único modelo que represente la gramática de atributos, pues cada vista de la gramática conforma un modelo por sí solo, compuesto a su vez, por otros modelos (estructurales o también denominados incontextuales, y semánticos) y cuyo contenido puede referir a elementos de modelos de otras vistas. Por otro lado, se requiere realizar transformaciones internas del modelo a fin de desambiguar sus términos y generar la gramática de atributos multivista final resultante de unificar todos los modelos. El procesador de documentos ya no se puede generar con herramientas de generación de traductores convencionales debido a que las gramáticas de atributos multivista son de ámbito general y requieren de un algoritmo de análisis que produzca bosques de análisis sintáctico correctos y adecuados a este tipo de gramáticas. Los algoritmos de análisis y construcción de bosques para gramáticas generales eficientes y aplicables existentes (sección 5.3.2.2) son de carácter reconocedor, y no se centran en esta correcta y adecuada construcción que deseamos, por lo que se ha tenido que desarrollar, implementar e integrar, el nuevo algoritmo MVGLR (sección 5.3.2.3) como componente núcleo del procesador de las aplicaciones de procesamiento XML (dicho algoritmo es equivalente a, e intercambiable por, el algoritmo MVLR). Para dar soporte a documentos XML, el entorno de ejecución de XLOP3 realiza la interconexión del analizador sintáctico multivista MVGLR configurado como traductor mediante un componente constructor de bosques atribuidos, con el analizador léxico XML configurado con SAX (sección 5.3.3.1). Este analizador léxico, junto con el módulo constructor de autómatas LALR, son los únicos componentes que han podido reutilizar, tras su adaptación y refinamiento, desde la arquitectura XLOP 1.0 a la nueva arquitectura XLOP3. En la Figura 5.5.14 se muestra la primera parte de la arquitectura XLOP3 referente a la creación del modelo de la gramática de atributos multivista final y, en la Figura 5.5.15 se muestra la segunda parte referente a la generación de la aplicación de procesamiento XML.

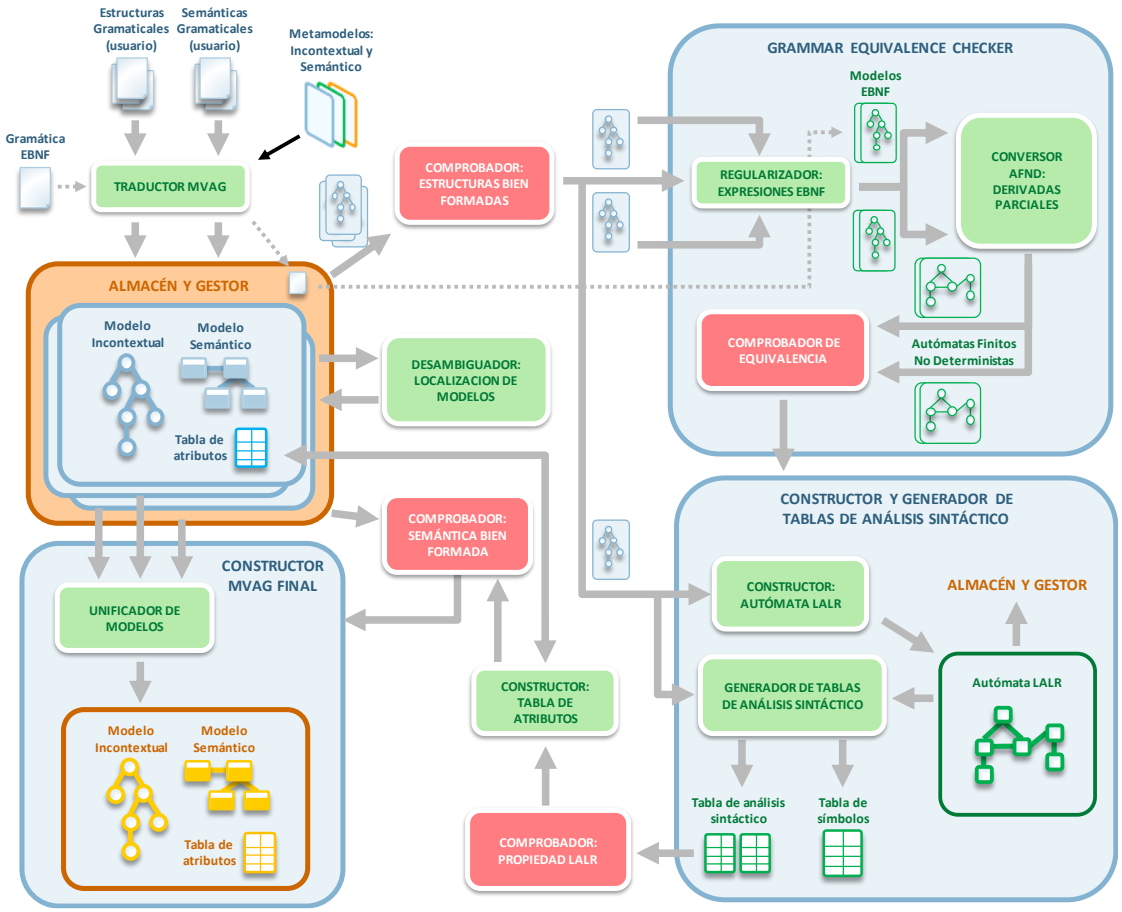


Figura 5.5.14. Arquitectura de XLOP3: obtención de la gramática de atributos multivista final.

La Figura 5.5.14 muestra el proceso de obtención del modelo final de gramática de atributos multivista (amarillo). Este proceso parte de los archivos de especificación multivista diferenciados en: gramáticas EBNF de base, estructuras gramaticales y, semánticas gramaticales asociadas a estas estructuras. El proceso integra los siguientes componentes:

- *Traductor MVAG* (Traductor Multi-View Attribute Grammar). Este componente es un traductor para el lenguaje de especificaciones multivista XLOP3 construido y generado utilizando la herramienta JavaCC. El traductor transforma las especificaciones modulares estructurales (estructuras gramaticales) en modelos incontextuales y, las especificaciones modulares semánticas (semánticas gramaticales) en modelos semánticos. Si se aporta, la gramática EBNF es transformada en un modelo EBNF. La creación de los anteriores modelos es regida por el metamodelo incontextual de XLOP3 que representa una gramática incontextual BNF o EBNF, y el metamodelo semántico de XLOP3 que representa las ecuaciones semánticas asociadas a una instancia de la estructura BNF (bibliotecas azul-verde-amarilla). Los modelos construidos son almacenados y gestionados por el *Almacén y Gestor*, clase que permite asociar los modelos semánticos con sus

respectivas estructuras gramaticales, entre otras gestiones. El traductor, así mismo, registra los errores sintácticos que puedan existir en las especificaciones gramaticales. Si existen errores, el proceso principal no continúa mientras no sean corregidos.

- *Desambiguador*: *localización de modelos*. Este componente dota de identificadores únicos a los elementos de las estructuras gramaticales y a sus semánticas gramaticales asociadas. La finalidad es impedir la mezcla incorrecta de elementos con nombres idénticos, pertenecientes a niveles locales de especificación o que provienen de diferentes modelos. Esta desambiguación, que se ha detallado en la sección 5.5.3.2, es necesaria para el módulo comprobador de semántica bien formada y para la unificación de los modelos en el modelo final de gramática de atributos multivista.

- *Comprobador: estructuras bien formadas*. Este componente realiza sobre cada modelo de estructura gramatical (modelo incontextual) una serie de comprobaciones para garantizar que dicha estructura está bien formada, comprobaciones que se han detallado en la sección 5.5.2.2. El comprobador registrará los errores e inconsistencias que detecte, y el proceso principal no continuará hasta que sean resueltos.

- *Grammar Equivalence Checker*. Este componente realiza la comprobación de conformidad automatizada entre dos modelos incontextuales. Si el resultado de la comprobación de la conformidad entre estructuras gramaticales no tiene éxito, la continuidad del proceso principal quedará suspendida. Este componente incluye, así mismo, los siguientes subcomponentes:

    - *Regularizador: expresiones EBNF*. Este componente transforma una gramática BNF en una gramática EBNF equivalente. Su comportamiento varía dependiendo de si se ha proporcionado o no una gramática EBNF de base. En caso afirmativo, aplica sobre cada subgramática asociada a cada símbolo de núcleo el método de traducción a expresiones regulares desarrollado en el Capítulo 4. En caso negativo, utiliza el método generalizado de transformación de una gramática BNF en una gramática EBNF desarrollado en dicho capítulo.

    - *Conversor AFND: derivadas parciales*. Utiliza el algoritmo de derivadas parciales (sección 4.3.4.4) para transformar las expresiones regulares implicadas en los modelos EBNF en autómatas finitos no deterministas. Dicha transformación opera de manera perezosa: sólo se transforman en autómatas las expresiones que son requeridas por el comprobador de la equivalencia. Así mismo, los AFNDs generados operan también perezosamente, expandiendo sus estados y transiciones conforme estos son requeridos.

  - *Comprobador de equivalencia*. Realiza la comprobación de equivalencia entre los autómatas anteriormente obtenidos, empleando el método descrito en la sección 4.5.4 y aplicando también, de manera perezosa, la determinización de los autómatas con el método de construcción por subconjuntos (sección 4.3.3).

- *Constructor LALR y generador de tablas de análisis sintáctico*. Construye el autómata LALR(1) para cada uno de los modelos incontextuales extraídos de la especificación, con el propósito de comprobar que cada una de estas gramáticas sea LALR(1). Posteriormente, se almacenan, y se genera las tablas de análisis sintáctico y tabla de símbolos (esta tabla asocia nombres simbólicos con identificadores numéricos). Los subcomponentes que forman este constructor y generador son:

  - *Constructor: autómata LALR*. A partir del modelo incontextual, realiza la construcción del correspondiente autómata LALR(1). Dicho autómata puede volcarse a archivo en formato VCG para poder ser visualizado mediante una herramienta externa como [yComp], por el entorno de depuración XLOP3 de la aplicación final generada.

  - *Generador de tablas de análisis sintáctico y tabla de símbolos.* Genera la tabla de símbolos a partir de los modelos incontextuales, tabla que alberga la traducción de los nombres de los elementos de los modelos (terminales, no terminales, producciones) a identificadores numéricos unívocos. Estos identificadores son referenciados en la tabla de análisis sintáctico obtenida de analizar los estados del autómata LALR(1) correspondiente.

  - *Comprobador: propiedad* LALR(1). Analiza la tabla de análisis sintáctico en busca de conflictos o múltiples acciones asociadas a una celda de su tabla acción. El hecho de presentar la tabla de análisis conflictos en una o más celdas implica que la propiedad LALR(1) del autómata no se preserva. Aunque no se cumpla esta propiedad para una estructura gramatical, se permitirá continuar el proceso principal si así se desea, con el aviso de que el procesador de documentos XML que se generará podrá funcionar de una manera que no garantiza la correcta y previsible evaluación de los árboles sintácticos atribuidos (sección 5.4).

- *Constructor: tabla de atributos*. Construye la tabla de atributos (azul) asociada a cada modelo semántico y su respectivo modelo incontextual. La tabla construida se almacena en el *Almacén y Gestor* junto con sus correspondientes modelos. El constructor también detecta y notifica inconsistencias de los atributos semánticos, impidiendo que el proceso principal prosiga mientras permanezcan.

- *Comprobador: semántica bien formada*. Junto con las tablas de atributos, se realiza a cada modelo de semántica gramatical (modelo semántico) una serie de comprobaciones para garantizar que las ecuaciones semánticas asociadas a su estructura gramatical están bien formadas. Las características que debe poseer dicha semántica se detalla en la sección 5.5.2.3. El comprobador registrará los errores que detecte y el proceso principal no continuará hasta que sean corregidos.

- *Constructor MVAG final*. Constructor de la gramática de atributos multivista final. Realiza la unificación inteligente de todos los modelos incontextuales y sus modelos semánticos asociados presentes en el *Almacén y Gestor*. El resultado es el modelo final de gramática de atributos multivista: modelo incontextual (amarillo) y modelo semántico (amarillo). La tabla de atributos final es fruto de unificar cada tabla de atributos local a dichos modelos (amarillo). Este proceso de unificación es llevado a cabo por el componente *Unificador de modelos*.

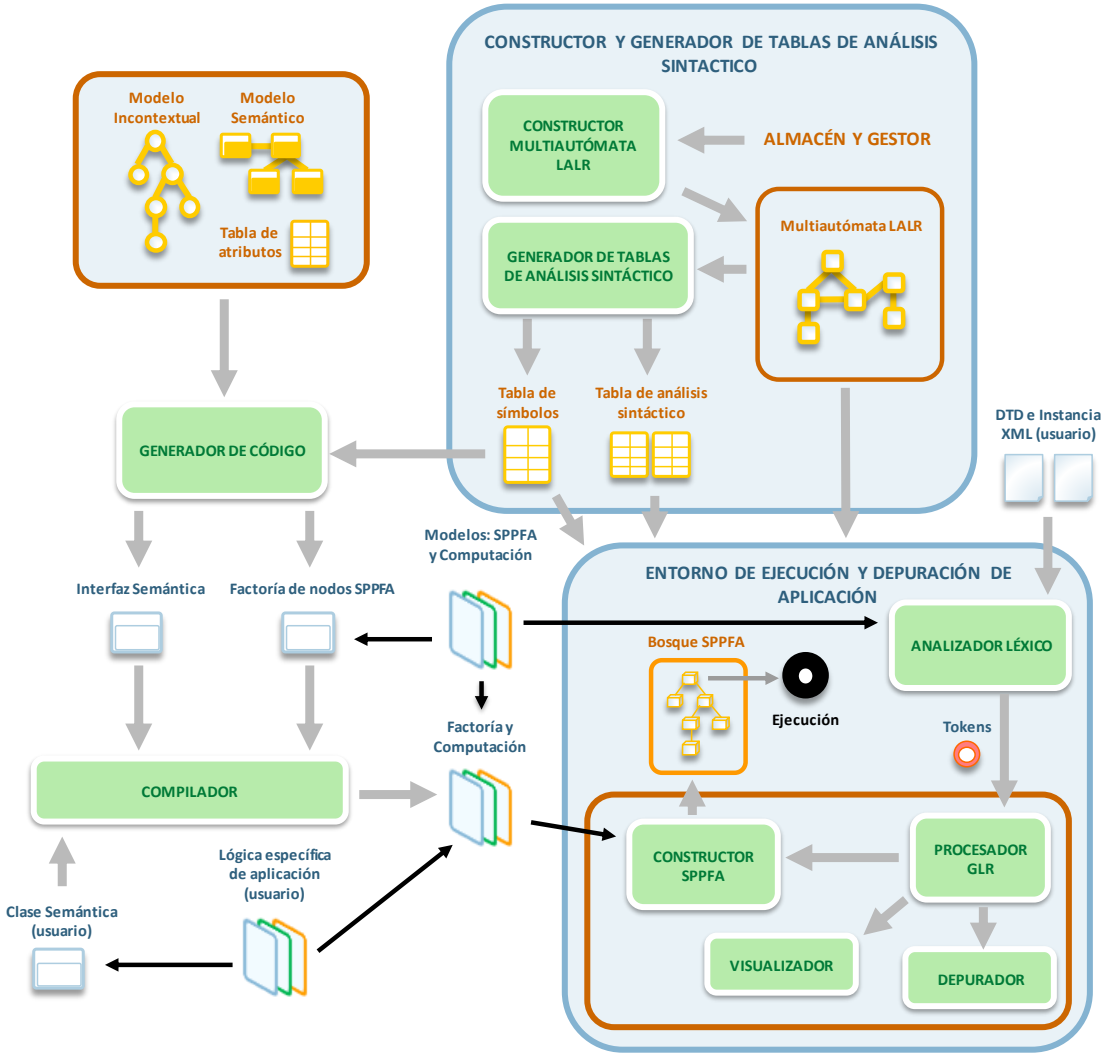


Figura 5.5.15. Arquitectura de XLOP3: generación de la aplicación de procesamiento XML.

La segunda parte de la arquitectura de XLOP3 se centra en la generación de la aplicación de procesamiento XML y la configuración de su procesador interno GLR presente en su entorno de ejecución y depuración generado por XLOP3. La Figura 5.5.15 muestra la arquitectura del proceso de generación de la aplicación de procesamiento XML construida con XLOP3, y cómo es la arquitectura del entorno de ejecución y depuración de la aplicación final. Partiendo de los autómatas LALR(1) individuales de cada gramática, se construye el multiautómata LALR(1) (amarillo) mediante el *Constructor y generador de tablas de análisis sintáctico*. Mientras que el autómata se vuelca a archivo por cuestiones de depuración (para el entorno de depuración de la aplicación, sección 5.5.4), la tabla de símbolos y tabla de análisis sintáctico que se obtienen de dicho autómata (amarillo) serán elementos clave para el *Procesador GLR* del entorno de ejecución. Aunque suponemos que la conjetura enunciada en la sección 5.3, que supone que la ausencia de conflictos en los autómatas LALR(1) individuales asegura la ausencia de conflictos en el correspondiente multiautómata, se cumple, en este componente comprobamos de nuevo la ausencia de conflictos, y prevemos la emisión de un aviso en caso de que, finalmente, el sistema descubra que la conjetura es falsa.

Los elementos necesarios para el funcionamiento correcto del *Entorno de ejecución y depuración* son los siguientes:

- *Tabla de símbolos*. Tabla de símbolos de la gramática de atributos multivista final, obtenida del autómata final por el módulo *Generador de tablas de análisis sintáctico*.

- *Tabla de análisis sintáctico*. Tabla de análisis sintáctico final de la gramática de atributos multivista final, obtenida del multiautómata final por el módulo *Generador de tablas de análisis sintáctico*.

- Biblioteca de *Modelos: SPPFA y Computación*. Contiene el metamodelo SPPFA que rige la construcción de un bosque de análisis sintáctico atribuido SPPFA (sección 5.3.3.2) y el modelo de gestor del valor y computación (sección 5.4.1), que permite la gestión y cálculo del valor de los atributos asociados a los nodos del bosque de análisis sintáctico.

- Bibliotecas de *Factoría y Computación*. Formadas por la *Clase Semántica* aportada por el usuario (sección 5.4.1) y su *Interfaz Semántica* (sección 5.5.3.1) generada por el *Generador de código* a partir del modelo incontextual, semántico y la tabla de atributos final. Los métodos de la clase semántica son los requeridos en los procesos de cómputo de los nodos construidos por la factoría de nodos. Estas bibliotecas contienen la implementación específica de la *factoría de nodos SPPFA* (sección 5.3.3.3) resultante de procesar la semántica de la gramática de atributos multivista final, elemento que es esencial para poder realizar la construcción de bosques de análisis sintáctico atribuidos evaluables (sección 5.4.2). Estas bibliotecas encapsulan la interconexión entre la capa lingüística y la capa lógica de

la aplicación, requiriendo los modelos de las bibliotecas *Modelos: SPPFA y Computación* y *Lógica específica de aplicación*.

Como puede observarse en la Figura 5.5.15, las bibliotecas *Factoría y Computación* requieren elementos que deben, por una parte, ser generados por el entorno XLOP3, y por otra parte ser aportados por el usuario. En concreto, el usuario debe aportar:

- Bibliotecas de *Lógica específica de la aplicación*. Es la lógica específica de la aplicación: marco, bibliotecas y/o recursos de la aplicación, que deben ser aportados por el usuario.

- *Clase Semántica*. Clase que implementa los métodos de la *Interfaz Semántica* (sección 5.5.3.1) generada por el *Generador de código*. Estos métodos son los métodos referidos por las funciones semánticas de los modelos semánticos. A través de los métodos de esta clase se consigue interconectar las tareas de procesamiento dirigidas por la sintaxis del lenguaje XML descritas en las gramáticas de atributos multivista (es decir, la capa lingüística) con el marco de aplicación, (la capa lógica).

Los elementos generados por el entorno XLOP3 se producen mediante el módulo *Generador de código*. Este módulo toma como entrada la tabla de símbolos final (amarillo), el modelo incontextual y semántico final (amarillo), y la tabla de atributos final (amarillo). Como resultado, produce el código de la implementación específica de la *Factoría de nodos* de la gramática de atributos multivista (sección 5.3.3.3) y la *Interfaz Semántica* (sección 5.5.3.1) cuyos métodos deben implementar la *Clase Semántica* aportada por el usuario. El modelo de construcción que utiliza la *Factoría de nodos* se rige por el modelo de la biblioteca *Modelos: SPPFA y Computación*. Tanto la clase semántica del usuario, la interfaz semántica y la factoría de nodos, son elementos de código que deben ser compilados por el módulo *Compilador* para generar las bibliotecas *Factoría y Computación*.

Por último, la aplicación final resultante integrará el *Entorno de ejecución y depuración*. Los elementos que conforman este entorno son aportados por el entorno XLOP3, y cuya configuración específica a través de las bibliotecas *Factoría y Computación*, *Tabla de símbolos* final, y *Tabla de análisis sintáctico* final, resultan en un procesador de documentos específico para la aplicación final. El proceso que se realiza interiormente en el entorno de ejecución y depuración, se inicia cuando se aporta la instancia documental XML junto con su DTD (si así lo requiere la instancia XML). En la Figura 5.5.15 se muestran estos elementos como *DTD e Instancia XML*. Consecuentemente, el entorno puede operar de manera directa o en modo de depuración (sección 5.5.4), comenzando por la inicialización del analizador léxico. Los componentes involucrados son:

- *Analizador léxico*. Adaptación del marco de procesamiento orientado a eventos SAX (sección 2.3.3.2) para funcionar como un analizador léxico que procesa la *DTD e Instancia XML*, devolviendo cada uno de los elementos de la instancia documental como *Tokens* para el *Procesador GLR*. Esta configuración se detalla en la sección 5.3.3.1. El contenido textual de un elemento XML #pcdata, y los atributos de las etiquetas de apertura, se incluirán en el contenido de su correspondiente token generado siguiendo la estructura designada para atributos léxicos definida en las bibliotecas *Modelos: SPPFA y Computación*.

- *Procesador GLR*. Es el componente principal que dirige y realiza el procesamiento de los documentos XML. Contiene la implementación de MVGLR (sección 5.3.2.3) (aunque también puede configurarse para funcionar con el MVLR o con la adaptación heurística del GLR). Realiza las peticiones de tokens al *Analizador léxico* y los procesa secuencialmente de uno en uno. Funciona como un reconocedor multivista que integra componentes que proporcionan valor añadido al mismo:

    - *Depurador*. Registra en detalle las acciones llevadas a cabo por el procesador GLR y su funcionamiento interno estructural (sección 5.5.4.2).

    - *Visualizador*. Construye el bosque SPPF resultante de procesar el documento XML cuyo volcado a formato VCG permite su visualizado gráficamente mediante la herramienta externa [yComp].

    - *Constructor SPPFA*. Construye el *Bosque SPPFA* (sección 5.3.3.2) Este es el bosque atribuido con procesos de cómputo SPPFA, resultante de procesar el documento XML de entrada, cuya construcción se realiza mediante peticiones a la factoría de nodos SPPFA (sección 5.3.3.3) de las bibliotecas de *Factoría y Computación*.

Una vez obtenido el *Bosque SPPFA*, la *Ejecución* efectiva de las tareas de procesamiento definidas para el procesador XML se realizan a través de dicho bosque SPPFA, iniciando su proceso de evaluación (sección 5.4.2). En esta evaluación, los procesos de cómputo asociados a los distintos nodos del bosque (cómputos definidos en la gramática de atributos multivista e interconectados con la lógica específica de aplicación a través de la clase semántica) se ejecutan realizando cada una de las tareas de procesamiento especificadas en cada una de las vistas, de la gramática de atributos multivista, mediante un único análisis y procesamiento del documento XML de entrada.

5.6 Caso de estudio: <eMU>

<eMU> (*educational technology-Move-Use*) es un sistema para el desarrollo de juegos interactivos con carácter educativo, que, en la línea de propuestas como [Moreno-Ger et al. 2007, Moreno-Ger et al. 2008], permite generar dichos juegos a partir de documentos XML. Un juego de <eMU> se compone de habitaciones con elementos interactivos en su interior, mostrados desde una perspectiva de cámara aérea en dos dimensiones. El usuario o jugador toma papel en el juego a través de un personaje que es capaz de moverse por el interior de la habitación e interactuar y experimentar con los elementos presentes en ella. La finalidad en el juego es lograr una serie de objetivos propuestos.

El sistema <eMU> integra un *marco de aplicación* <eMU>, constituido por los siguientes componentes:

- Un *motor de videojuegos* experimental denominado <MUe> (*Move-Use-engine*). Este motor consiste en un conjunto de bibliotecas de desarrollo para el modelado y representación visual de los escenarios y elementos de juegos, y también aporta mecánicas básicas para este tipo de juegos.

- Un *marco de juegos* <eMU> que caracteriza el tipo de juegos educativos concreto que puede desarrollarse con el sistema.

El sistema integra, así mismo, un generador *de juegos* <eMU>. Dicho generador permite procesar descripciones de juegos <eMU> mediante documentos XML, y realiza las instanciaciones necesarias en el marco de aplicación <eMU> para generar de manera efectiva los juegos descritos. De esta forma, la descripción de un juego de <eMU> se realizará mediante un documento estructurado con un lenguaje de marcado descriptivo XML de alto nivel, cuyos contenidos puedan ser fácilmente elaborados y modificados sin requerir altos niveles de conocimientos informáticos. La información que albergarán estos documentos XML permitirá la creación en su totalidad, de una experiencia interactiva diferente para cada instancia documental.

Debido a que la propuesta surge de una idea experimental, y como toda aplicación en desarrollo, tanto el motor como el marco de aplicación sufrirán continuos cambios a lo largo del tiempo, estos cambios continuos no sólo afectarán al marco de aplicación, sino también podrá afectar a la lógica de creación y a la gramática documental descriptiva de estos juegos. De la misma manera, estos cambios podrán producirse en dirección inversa, como consecuencia del enriquecimiento de <eMU> y de su gramática documental para dar cabida a nuevas necesidades expresivas. Por tanto, es esencial optar por una forma de desarrollo práctica y sostenible, que, a su vez, posibilite un procesamiento eficiente de los documentos. Para ello, para llevar a cabo el desarrollo del generador <eMU>, puede emplearse el enfoque de desarrollo de procesadores XML dirigidos por la sintaxis con el entorno XLOP3 y gramáticas de atributos multivista. Gracias al tipo de especificaciones soportado por XLOP3 se podrán describir múltiples tareas de procesamiento (creación de elementos, interconexión entre

elementos, adición de eventos y comportamientos) de manera independiente, obteniendo como resultado un procesador de documentos eficiente, capaz de realizar todas las tareas mediante un único análisis del documento de entrada.

De esta forma, y como complemento a los casos relativamente sencillos desarrollados anteriormente a fin de amenizar la exposición y explicación del enfoque metalingüístico de esta tesis, mediante la elaboración de este caso de estudio, de sustancial complejidad, cuya problemática atiende a escenarios que suceden de manera común en el desarrollo de videojuegos (generación de escenarios de juego a partir de documentos), se demostrará además la viabilidad real y la potencia de desarrollo que brinda dicho enfoque metalingüístico.

En las secciones 5.6.1 a 5.6.3 se detallará la construcción de la aplicación de procesamiento XML <eMU> con XLOP3 mediante gramáticas de atributos multivista, siguiendo la metodología de desarrollo descrita en la sección 5.2. Para finalizar, se presentará el resultado obtenido en la sección 0.

5.6.1 Creación de juegos con <eMU>

La aplicación de procesamiento XML <eMU> se concibe como una herramienta que permite generar juegos educativos a partir de documentos XML. Su ámbito de uso está enfocado a un público de temprana edad, y su ámbito de aplicación a lugares de enseñanza con recursos informáticos de bajo coste (equipos de limitada potencia). Siguiendo ese propósito, <eMU> se adhiere a un sencillo diseño, un videojuego en dos dimensiones, pero que, sin embargo, sea capaz de aportar una rica experiencia educativa a base de simples interacciones. Sus escenarios, contenidos, misiones e interacciones pueden crearse con diversidad y describirse fácilmente en documentos estructurados. Las siglas <eMU> atienden a *educational technology-Move-Use*, lo que se traduce en la simplicidad de acciones de entrada (mover y usar) que el usuario necesitará realizar para disfrutar y completar un juego. El jugador de los juegos generados con <eMU> toma protagonismo a través de un personaje que puede moverse por diferentes escenarios (habitaciones vistas desde una perspectiva aérea) verticalmente y horizontalmente por cuadrículas, e interactuar con los elementos presentes en ellas. La finalidad consiste en completar unos determinados objetivos, presentados como misiones y tareas de misión. Estos juegos pueden ser descritos en instancias documentales XML, donde se describen cada una de las habitaciones, elementos, misiones, tareas, y comportamientos de las interacciones. Para dar una mayor diversidad y profundidad a la experiencia de juego, el jugador dispondrá de un inventario que le permitirá almacenar elementos del escenario. De esta manera, se introduce un sistema de eventos que dará lugar a la creación de un sinfín de situaciones diferentes en función de qué acciones realice el jugador con los elementos del escenario que interactúa y el estado de su inventario, e incluso el estado de los elementos y/o de los objetivos de misión alcanzados.

En la sección 5.6.1.1 se describe el tipo de experiencias de juego que debe ofrecer <eMU>, ilustrándolo con un ejemplo de juego educativo denominado *Fruits and Colors*. La gramática documental XML utilizada en <eMU> para describir los juegos se detalla en la sección 5.6.1.2.

5.6.1.1 Escenarios de juego y funcionalidades

Describiremos el tipo de juegos y experiencias que ofrece un juego típico de <eMU> con el desarrollo de un ejemplo: el juego *Fruits and Colors*. La Figura 5.6.1 muestra el diseño esquemático de las habitaciones, elementos y eventos del juego. El objetivo educativo de este juego consiste en enseñar inglés, estimulando al usuario a que explore los elementos del escenario y asocie sus descripciones textuales con el contenido visual interactivo. El jugador tomará el papel de un niño que deberá ayudar a una niña a recuperarse. Concretamente, la misión principal del juego consiste en la elección de un tipo de fruta descrita por la niña, por su nombre y color característico, presentándose al usuario, en una de las habitaciones, varias frutas diferentes a escoger, de las cuales sólo una conlleva a una solución correcta. Para ello, el juego contará con cuatro habitaciones: pasillo (*Corridor*), cuarto de la chica (*Girl's Room*), cocina (*Kitchen*), y una habitación cerrada (*Closed Room*). En estas habitaciones, el jugador podrá mover el personaje (*Player*) desde la habitación pasillo y transitar a otras habitaciones a través de elementos puerta (*Door*). Posteriormente, el jugador entablará diálogo con la chica (*Girl*) que se encuentra en el cuarto de la chica (*Girl's Room*), y a través de esta interacción se establecerá una serie de tareas para completar la misión principal del juego: preparar un zumo de naranja y dárselo a la chica. Para completar esta misión, será necesario hacer uso del inventario del jugador para recoger los elementos fruta (*Banana, Orange, Apple, Pear*) de la habitación cocina (*Kitchen*), y utilizar estos elementos en el exprimidor (*Blender*) de esa misma habitación. Como resultado se obtendrá un zumo de fruta que, si es el correcto, al entregárselo a la chica, ésta entregará al jugador una llave (*Key*) para desbloquear la puerta de la habitación cerrada, cuyo elemento regalo (*Gift*) que presenta la habitación, completará la misión principal del juego.

Para que este juego sea funcional, <eMU> debe soportar las siguientes características y funcionalidades:

- Representar visualmente, en dos dimensiones y por cuadrículas, una habitación. La habitación podrá tener una imagen de fondo, unas dimensiones en ancho y alto en unidades de cuadrículas, y una posición que fije dónde se ubica relativamente respecto a la escena. La esquina superior izquierda se corresponderá a la posición (0,0), el valor del eje x se incrementará horizontalmente progresando hacia la derecha, y el valor del eje y se incrementará verticalmente progresando hacia abajo, hasta un máximo de 13x8 cuadrículas.

- Representar visualmente los elementos de una habitación. Los elementos serán representados por cuadrículas ubicadas en posiciones relativas a la habitación. Cada una de sus cuadrículas podrá poseer una imagen con transparencia, la cuales se podrán rotar/reflejar y ubicar localmente en una posición relativa a la posición del elemento en la habitación.

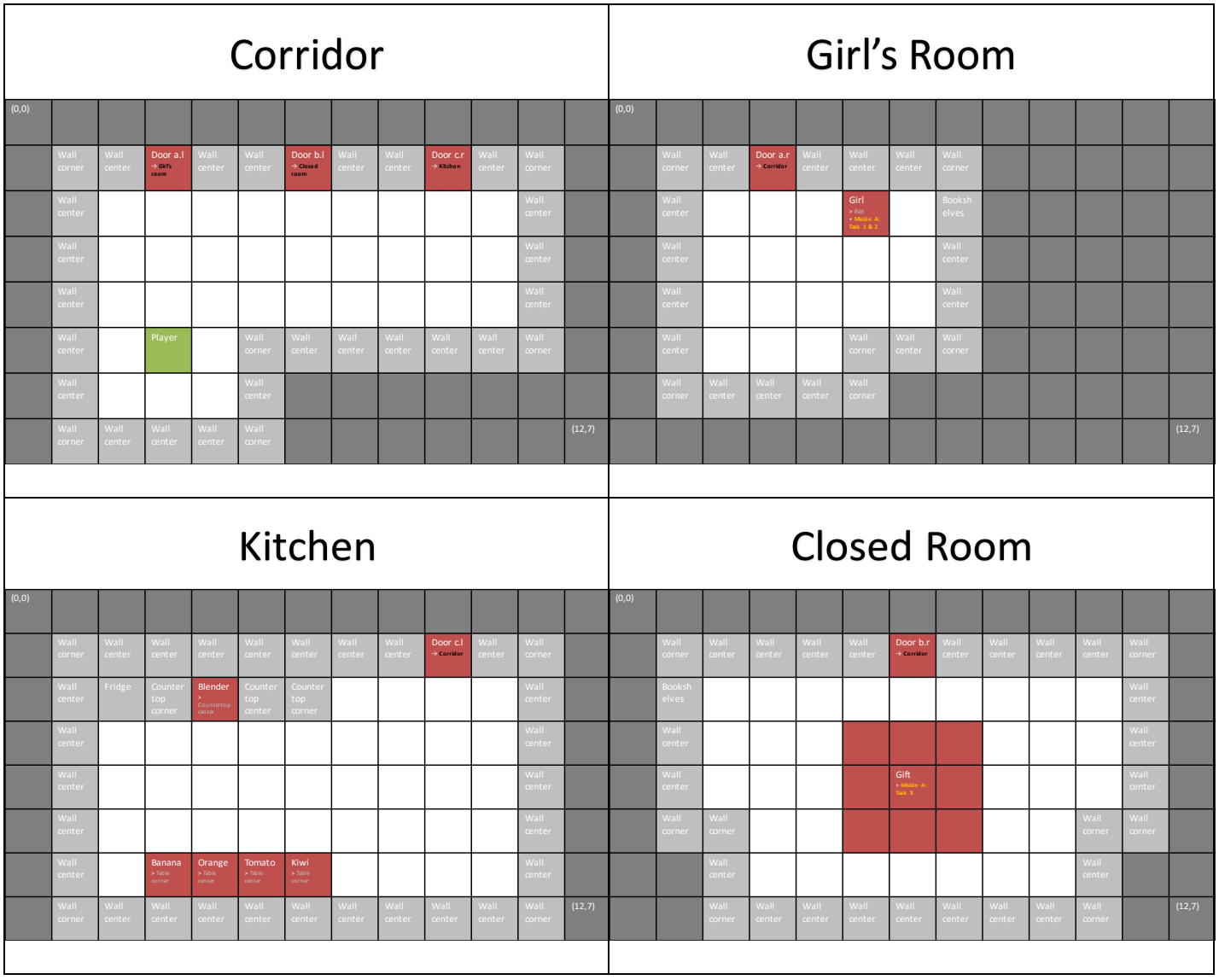


Figura 5.6.1. Diseño de habitaciones, elementos y eventos de *Fruits and Colors*.

- Existirán misiones compuestas por tareas. Las tareas tendrán estado de visible/invisible y completo/incompleto, y en función de dicho estado, podrán mostrarse al jugador de una u otra manera. Las misiones podrán ser ocultas, permitiendo utilizar sus tareas como una lógica añadida en la creación de eventos. Si una tarea es visible, su descripción se mostrará al usuario si su misión está seleccionada. El usuario podrá seleccionar entre distintas misiones para visualizar

sus tareas visibles. En la Figura 5.6.1 se representan las tareas que se completarán al interactuar con un elemento con el símbolo "+".

- Los elementos podrán ser de tres tipos diferentes: estáticos (gris claro), interactivos (rojo), o jugador (verde). Debido a que podrán ubicarse varios elementos en la misma posición, los elementos interactivos y el jugador se representarán siempre por encima de los elementos estáticos. Este orden viene representado en la Figura 5.6.1 con el signo ">" mostrando prioridad de representación.

- Los elementos podrán tener más de un estado que permitirá cambiar su descripción, visualización, u otros aspectos. De esta manera, por ejemplo, el elemento jugador podrá visualizarse de diferente manera al cambiar de dirección, o cada tipo de fruta cambiar a un estado zumo (por ejemplo, *Orange* puede cambiar a estado *Orange Juice*).

- El elemento jugador será controlado por el usuario, teniendo la capacidad de poder cambiar de dirección y desplazarse. Este desplazamiento se realizará una casilla consecutiva en la dirección en la que mira el elemento jugador, sólo si dicha casilla no posee un elemento no transitable. En principio, todos los elementos presentes en la Figura 5.6.1, tanto interactivos como estáticos, no son transitables. De hecho, existen muchos elementos estáticos que son simplemente muros para que el jugador no pueda salirse de la habitación en la que se encuentra.

- El jugador deberá tener una acción de interacción con los elementos de las habitaciones y poseer un inventario de elementos. De su inventario, podrá seleccionar únicamente un elemento a la vez, consultar la descripción de dicho elemento, y podrá utilizarlo mediante su acción de interacción sobre otro elemento (o ninguno).

- Los elementos se podrán mover entre habitaciones, pero siempre bajo el concepto de conservación de la materia: no podrán crearse, duplicarse ni destruirse. Sólo podrán ser ocultados al jugador de dos maneras diferentes: ubicándolos en lugares no visibles a éste o estableciéndolos como ocultos. En el esquema de la Figura 5.6.1 no se muestran los elementos estáticos adicionales, pues estos deben existir en posiciones fuera del escenario para estar ocultos (por ejemplo, en la coordenada (-10,-10) de la habitación *Girl's Room*, la llave *Key*.

- Los elementos deberán poseer una descripción que se mostrará al jugador si el elemento jugador los mira directamente estando en una casilla consecutiva. Esta realimentación es esencial para fomentar el aprendizaje y asociación de nombres y conceptos con sus respectivos elementos visuales.

- Los elementos interactivos/jugador podrán poseer eventos. Los eventos serán comportamientos añadidos a estos elementos, que se realizarán si se cumple una serie de condiciones establecidas. Por ejemplo, el exprimidor (*Blender*) "producirá"

un elemento zumo de naranja (*Orange Juice*) sólo si el jugador tiene seleccionado en su inventario un elemento naranja (*Orange*) e interactúa con el exprimidor (realmente cambia el estado del elemento interactivo, pues muestra un mensaje en su recogida, de *Orange* a *Orange Juice*). También, mediante estos eventos, será posible cambiar de habitación activa al interactuar el jugador con una puerta, representado en la Figura 5.6.1 mediante el símbolo "→" bajo el nombre de dichas puertas como elemento interactivo.

- La condición de los eventos será una expresión booleana con asociación (izquierda a derecha) y precedencia de operadores (*NOT* > *AND* > *OR*). Las condiciones serán preguntas de uno de los tipos siguientes, con respuesta cierto o falso:
    - ¿El inventario contiene el elemento X? / ¿El elemento X está seleccionado en el inventario?
    - ¿La tarea X de la misión Y está visible/completa?
    - ¿El elemento X está en el estado Y?
- Los eventos, en caso de éxito, realizarán instrucciones consecutivas de los tipos siguientes:
    - Mostrar un mensaje al usuario.
    - Establecer como visible/invisible o completa/incompleta una tarea de una misión. La misión en sí se considerará completa si todas sus tareas se han completado.
    - Mover un elemento de posición, de dirección y de habitación.
    - Recoger un elemento de la habitación activa y almacenarlo en el inventario o dejar un elemento del inventario en una ubicación dentro de la habitación activa.
    - Cambiar el estado de un elemento de la habitación (incluye el jugador) o del inventario (o directamente el elemento seleccionado en éste).
- La ejecución de los eventos se realizará tras realizar la evaluación de su expresión condicional. Los eventos se evaluarán consecutivamente. Esta evaluación se iniciará de forma pasiva y reiterada, o de manera activa y disparada cuando el elemento jugador interactúe con un elemento interactivo de la habitación (en esta última forma se ejecutan los eventos en orden y si uno es exitoso no se continúa con la ejecución de los restantes). De esta manera, podrán configurarse eventos que no dependan de interacciones sino de estados del juego (a través del estado de las tareas de una misión, de algún elemento de la habitación activa, e incluso según la

existencia de un elemento en el inventario). También mediante eventos pasivos se podrá establecer las misiones y tareas inicialmente activas.

- Las misiones estarán formadas por tareas en estado incompleto e invisible al inicio. Una misión será visible si contiene al menos una tarea en estado visible. La misión aparecerá como completa si todas sus tareas se establecen como completas. El usuario podrá seleccionar una misión disponible y, como consecuencia, deberá mostrarse al usuario la descripción de sus tareas visibles. Se podrán completar tareas sin establecerlas visibles en ningún momento, otorgando así la capacidad de añadir lógica oculta adicional para realizar eventos o comportamientos más sofisticados.

El soporte básico de estas funciones se plasmará en el desarrollo del marco de la aplicación <eMU> (sección 5.6.2), dejando los aspectos constructivos de los escenarios de juego dirigido por el procesamiento de la sintaxis de los documentos XML, procesamiento que se especificará mediante gramáticas de atributos multivista (sección 5.6.3). La sintaxis, el lenguaje, de los documentos XML, se describe en la sección 5.6.1.2.

5.6.1.2 Documentos de juego

La gramática documental modelizará una notación simple y suficiente para alojar debidamente los datos descriptivos de los elementos que forman un juego de <eMU>. La estructura se diseña de manera jerárquica, tal y cómo muestra la DTD de la Figura 5.6.2.

De esta forma:

- Un juego (<game>) se compone de misiones (<quest>) formadas por tareas (<task>), y habitaciones (<room>).

- Las habitaciones se componen de elementos de distintos tipos (estáticos: <static>, jugador: <player>, interactivos: <interactive>). Estos elementos tendrán estados (<state>) y, dependiendo de su tipo, eventos (<event>).

- Los eventos se forman mediante una expresión condicional (<conditionExp>) de operandos condicionales (<cond.inventory>, <cond.quest>, <cond.state>), y por instrucciones ejecutables (<ins.inventory>, <ins.quest>, <ins.moveTo>, <ins.message>, <ins.state>).

Cada uno de los elementos de juego dispondrá de características principales y secundarias. Las características principales aparecerán en los documentos mediante contenido textual (#pcdata) de etiquetas XML (<nameID>, <description>, etc.). Por su parte, las características secundarias, aquellas que no tienen que ser necesariamente especificadas por el diseñador del juego, sino que pueden tener un valor por defecto, vendrán dadas mediante atributos XML de etiquetas (por ejemplo, atributo *passable* de <state.attributes>).

eMU.dtd

```
<!ELEMENT game (quest+, room+)>
<!ELEMENT quest (nameID, description?, task+)>
<!ELEMENT task (#PCDATA)>

<!ELEMENT nameID (#PCDATA)>
<!ELEMENT description (#PCDATA)>

<!ENTITY % elemType '(player | static | interactive)'>
<!ELEMENT room (nameID, description?, room.attributes,
(%elemType;)*)>
<!ELEMENT room.attributes EMPTY>
<!ATTLIST room.attributes
  posX CDATA "0"
  posY CDATA "0"
  width CDATA "13"
  height CDATA "8"
  backgroundImage CDATA "imgs\black.png"
 >

<!ENTITY % elemBody '(nameID, elem.attributes, state+)'>
<!ELEMENT static (%elemBody;)>
<!ELEMENT interactive (%elemBody;, (event)*)>
<!ELEMENT player (%elemBody;, (event)*)>
<!ATTLIST player
  id CDATA "1"
>

<!ELEMENT elem.attributes (#PCDATA)>
<!ATTLIST elem.attributes
  posX CDATA "-99"
  posY CDATA "-99"
  lookTo CDATA "down"
  initState CDATA "default"
>

<!ELEMENT state (nameID, description?, state.attributes, tiles?) >
<!ELEMENT state.attributes EMPTY>
<!ATTLIST state.attributes
  passable CDATA "false"
  hidden CDATA "false"
>

<!ELEMENT tiles (tile*)>
<!ELEMENT tile (#PCDATA)>
<!ATTLIST tile
   posX CDATA "0"
   posY CDATA "0"
   mirror CDATA "false"
   lookTo CDATA "down"
>

<!ELEMENT event (description?, conditionExp?, actions)>
<!ATTLIST event
   trigger CDATA "use"
>
```

```
<!ENTITY % condType '(cond.inventory | cond.quest | cond.state)'>
<!ENTITY % c_opnd '(NOT | P | %condType;)'>
<!ENTITY % c_exp '(%c_opnd;,(OR?,%c_opnd;)*)'>
<!ELEMENT P (%c_exp;)>
<!ELEMENT NOT (%c_opnd;)>
<!ELEMENT OR EMPTY>
<!ELEMENT conditionExp (%c_exp;)>
<!ELEMENT cond.quest (#PCDATA)> <!-- question:isVisible/isComplete a
task of a quest. -->
<!ATTLIST cond.quest
   question CDATA #REQUIRED
   taskNumber CDATA #REQUIRED
>
<!ELEMENT cond.inventory (#PCDATA)> <!--
question:isSelected/contains an element in inventory. -->
<!ATTLIST cond.inventory
   question CDATA #REQUIRED
>
<!ELEMENT cond.state (#PCDATA)> <!--
question:isInRoom/isInInventory/selected if the actor is in the given
state. -->
<!ATTLIST cond.state
  question CDATA #REQUIRED
  actor CDATA "PLAYER"
>

<!ENTITY % insType '(ins.inventory | ins.quest | ins.room | ins.message
| ins.moveTo | ins.state)'>
<!ELEMENT actions (%insType;)+>
<!ELEMENT ins.quest (#PCDATA)> <!--
action:setVisible/setInvisible/setComplete/setIncomplete a task of a
quest. -->
<!ATTLIST ins.quest
   action CDATA #REQUIRED
   taskNumber CDATA #REQUIRED
>
<!ELEMENT ins.inventory (#PCDATA)> <!-- action:pick/place an element
from inventory to the room. -->
<!ATTLIST ins.inventory
   action CDATA #REQUIRED
   posX CDATA "-99"
   posY CDATA "-99"
>
<!ELEMENT ins.moveTo (#PCDATA)> <!-- Moves an actor to a (x,y)
position of a room. -->
<!ATTLIST ins.moveTo
   actor CDATA "PLAYER"
   lookTo CDATA "down"
   posX CDATA "3"
   posY CDATA "6"
>
<!ELEMENT ins.message (#PCDATA)> <!-- Prints a message. -->
<!ELEMENT ins.state (#PCDATA)> <!--
action:setInRoom/setInInventory/setSelected actor to new state. -->
<!ATTLIST ins.state
  action CDATA #REQUIRED
  actor CDATA "PLAYER"
>
```

Figura 5.6.2. Gramática documental (DTD) de <eMU>.

La Figura 5.6.3 muestra un fragmento del documento XML que recrea el escenario de juego *Fruits and Colors*. Como puede observarse, en la primera parte del documento se realiza la descripción de la misión principal (con nombre en clave *Mission A*) y de cada una de sus tareas (cuyo nombre en clave numérico será inferido según número de aparición en el documento). Posteriormente, se define la habitación con nombre en clave *Corridor*. En ella, residirá el elemento jugador <player>. Se espera, por contener el elemento jugador con identificador

numérico *id* con valor "1", que tras el procesamiento del documento se establezca como habitación inicial esta habitación *Corridor*. De manera más detallada puede observarse que:

FruitsAndColors.xml

```
<!DOCTYPE game SYSTEM 'eMU.dtd'>
<game>
  <quest>
    <nameID>Mission A</nameID>
    <description>Help at home.</description>
    <task>Enter the Girl's room and talk to the girl who is inside.</task>
<!--1-->
    <task>Prepare the fruit juice that the girl asked you, and give it to
her.</task> <!--2-->
    <task>Get your reward.</task> <!--3-->
  </quest>
  <quest>
    <nameID>Extra logic</nameID>
    <task>Use the key to open the Closed Room's door.</task> <!--1-->
    <task>Girl is in the Closed Room</task> <!--2-->
  </quest>

  <room>
    <nameID>Corridor</nameID>
    <description>House corridor.</description>
    <room.attributes backgroundImage="imgs\corridor_bg.png"/>

    <player id="1">
      <nameID>Little boy</nameID>
      <elem.attributes posX="3" posY="5" initState="Look up"/>
      <state>
        <nameID>Look up</nameID>
        <description>Player 1 looking up.</description>
        <state.attributes/>
        <tiles>
          <tile lookTo="up">imgs\little_boy.png</tile>
        </tiles>
      </state>
      <state>
        <nameID>Look down</nameID>
        <description>Player 1 looking down.</description>
        <state.attributes/>
        <tiles>
          <tile>imgs\little_boy.png</tile>
        </tiles>
      </state>
      <state>
        <nameID>Look left</nameID>
        <description>Player 1 looking left.</description>
        <state.attributes/>
        <tiles>
          <tile lookTo="left">imgs\little_boy.png</tile>
        </tiles>
      </state>
      <state>
        <nameID>Look right</nameID>
        <description>Player 1 looking right.</description>
        <state.attributes/>
        <tiles>
          <tile lookTo="right">imgs\little_boy.png</tile>
        </tiles>
      </state>
      <event trigger="repeat">
        <description>Setup Mission A</description>
        <actions>
          <ins.quest action="setVisible" taskNumber="1">Mission
A</ins.quest>
        </actions>
      </event>
    </player>
    <interactive>
      <nameID>Door b.l</nameID>
      <elem.attributes posX="6" posY="1" initState="Locked"/>
      <state>
        <nameID>default</nameID>
        <description>Closed room door.</description>
        <state.attributes/>
        <tiles>
          <tile>imgs\door_b.png</tile>
        </tiles>
      </state>
      <state>
        <nameID>Locked</nameID>
        <description>Closed room door (locked).</description>
        <state.attributes/>
        <tiles>
          <tile>imgs\door_b_closed.png</tile>
        </tiles>
      </state>
      <event>
        <description>Go to the Closed room.</description>
        <conditionExp>
          <cond.quest question="isComplete" taskNumber="1">Extra
logic</cond.quest>
        </conditionExp>
        <actions>
          <ins.moveTo posX="6" posY="2">Closed Room</ins.moveTo>
          <ins.message>You went to the closed room.</ins.message>
          <ins.state action="setInRoom" actor="Little boy">Look
down</ins.state>
        </actions>
      </event>
      <event>
        <description>Try to open the door without key.</description>
        <conditionExp>
          <cond.state question="isInRoom" actor="Door
b.l">Locked</cond.state>
          <NOT><cond.inventory
question="isSelected">Key</cond.inventory></NOT>
        </conditionExp>
        <actions>
          <ins.message>You need a key to open the door.</ins.message>
        </actions>
      </event>
      <event>
        <description>Open the door with the key.</description>
        <conditionExp>
          <cond.state question="isInRoom" actor="Door
b.l">Locked</cond.state>
          <cond.inventory question="isSelected">Key</cond.inventory>
        </conditionExp>
        <actions>
          <ins.message>You opened the Closed room.</ins.message>
          <ins.inventory action="place">Key</ins.inventory>
          <ins.state action="setInRoom" actor="Door
b.l">default</ins.state>
          <ins.quest action="setComplete" taskNumber="1">Extra
logic</ins.quest>
        </actions>
      </event>
    </interactive>
```

Figura 5.6.3. Extracto del documento XML del escenario de juego *Fruits and Colors* para <eMU>.

- El elemento jugador posee un evento que siempre se ejecutará (atributo de evento *trigger* con valor “repeat”) y que establecerá la misión/tarea de inicio del juego.
- Por otra parte, posee diversos estados, para poderse mostrar al jugador con una imagen diferente según la dirección en la que se mueva/mire.
- Otro elemento que contiene la habitación *Corridor* es de tipo interactivo, con nombre en clave *Door b.l*. Este elemento representa una puerta cerrada (estado inicialmente cerrado) que posee una serie de eventos con condiciones e instrucciones para: a) cambiar la escena por la habitación *Closed Room* y mover al jugador a ésta, b) mostrar un mensaje si la puerta está bloqueada, c) desbloquear la puerta con un elemento llave de nombre en clave *Key*. Como puede observarse, para ello se utilizan diversas instrucciones y maneras de llevarlo a cabo, comprobando objetivos realizados, estados de elementos y la selección en el inventario del jugador.

5.6.2 El marco de aplicación <eMU>

Como ya se ha indicado anteriormente, el marco de aplicación <eMU> está formado por dos componentes: un motor de videojuegos <MUe>, y un marco de juegos <eMU>, que acota el tipo de juegos educativos que se van a desplegar sobre dicho motor. A continuación, se describen ambos componentes.

5.6.2.1 El motor de videojuegos <MUe>

El motor de videojuegos <MUe> es un conjunto de bibliotecas que implementa la funcionalidad necesaria para realizar la presentación visual de los elementos de juego, aporta mecanismos de ejecución de lógica añadible a dichos elementos, y por defecto, una interfaz de interacción para el usuario con un mecanismo de entrada básico (para el manejo del personaje del usuario). El motor aporta un modelo de objetos elemental para modelizar los distintos elementos de un juego: misiones, tareas, habitaciones, elementos interactivos y estáticos, eventos e instrucciones, etc.

La Figura 5.6.4 muestra las clases e interfaces del motor <MUe> que permiten realizar las funciones principales de presentación visual y ejecución de lógica específica. De esta forma:

- Los distintos componentes que maneja el motor se caracterizan por poder ser representados visualmente extendiendo los métodos de la clase *DrawableA*, y por contener una lógica específica ejecutable a través de la implementación del método *logic* de la interfaz *LogicableI*.

- La clase *Executor* rige el sistema principal de ejecución y representación visual, y alberga una lista de ejecutores (implementan la clase abstracta *ExecA*) que se dibujarán invocando su método *draw* y se ejecutarán invocando su método *logic* muchas veces por segundo, de manera reiterada e indefinida.

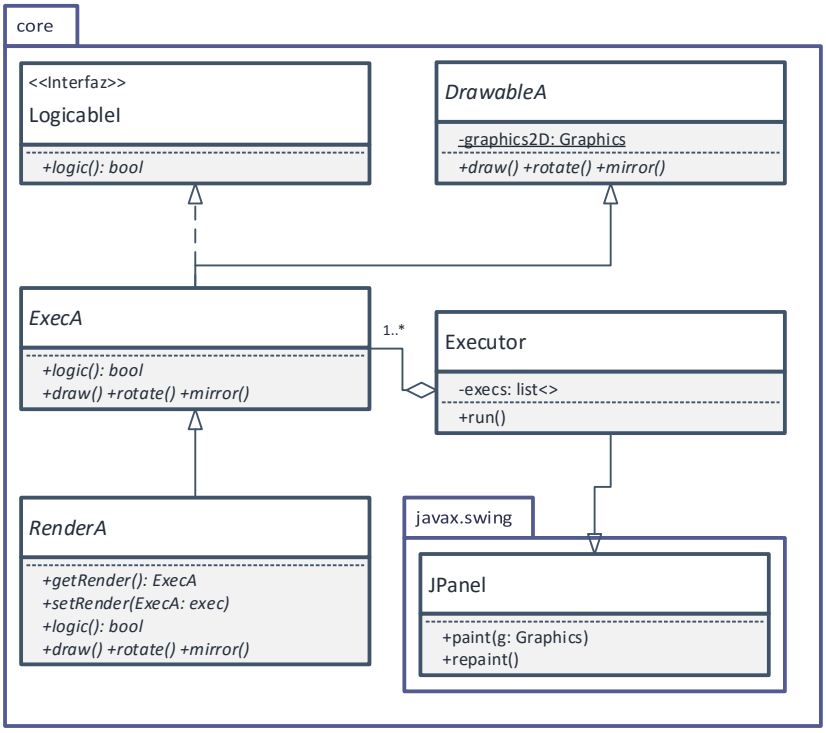


Figura 5.6.4. Clases e interfaces de presentación visual y ejecución de lógica específica de elementos de <MUe>.

- La clase abstracta *RenderA* es un ejecutor envoltorio de un ejecutor de tipo específico, cuyo propósito es que su ejecutor interno pueda cambiarse de manera indirecta, aportándose únicamente el envoltorio a la clase ejecutora *Executor*. De esta manera, mediante una implementación de *RenderA* que albergue como único ejecutor una habitación (instancia de la clase *Room*), establecida como habitación actual, se podrá cambiar la habitación actual que se ejecuta y visualiza en la escena de la aplicación <eMU> de manera indirecta. Esta implementación concreta de dicha clase abstracta se realizará, por tanto, en la lógica de soporte al generador.

La Figura 5.6.5 muestra las clases relativas a las habitaciones en <MUe> y a su estructura, en forma de los elementos que las pueblan. De esta forma:

- Una habitación (*Room*) pueden contener elementos (*Element*), que pueden adoptar diferentes estados (*State*).

- A su vez, el estado de un elemento puede implicar elementos más simples, que intervendrán en la configuración final del mismo, configuración que dependerá, por tanto, de los estados que adopta a distintos niveles de agregación.

De esta forma, la estructura de una habitación es jerárquica. En la raíz se encuentra la propia habitación. En el segundo nivel los elementos, que, a su vez, se pueden estructurar en términos de otros elementos más simples (compuestos, *macrobloques*, o simples, *bloques*). De esta forma, la habitación será la encargada de iniciar el proceso de visualización, que operará de arriba abajo, de acuerdo con la estructura jerárquica.

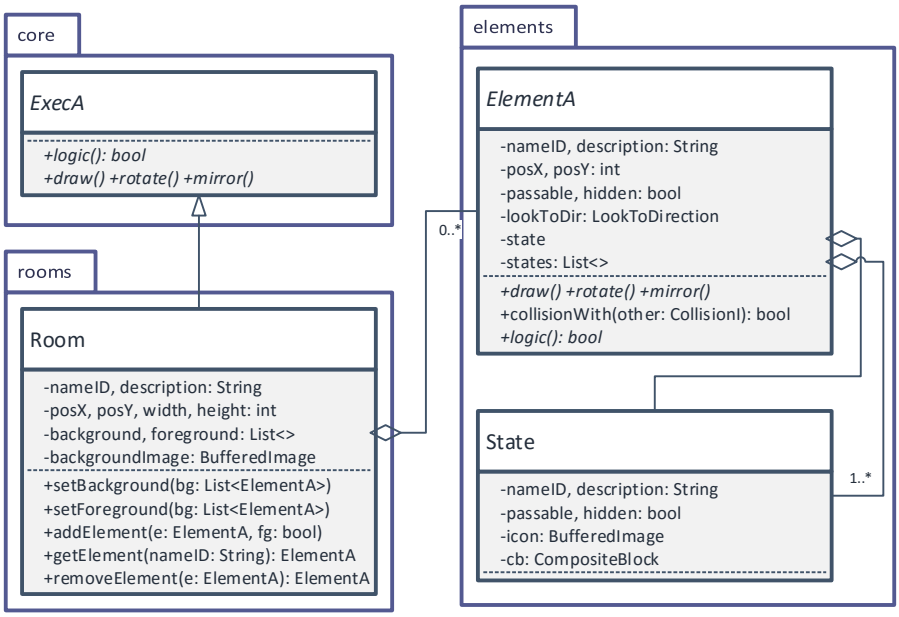


Figura 5.6.5. Modelo de objetos de tipo habitación en <MUe>.

La Figura 5.6.6 esboza los distintos tipos de elementos (estáticos, *StaticElement* e interactivos, *InteractiveElement*) que pueden poblar dichas habitaciones. A bajo nivel, los elementos poseen un sistema de colisiones para que el elemento jugador no pueda traspasar ciertos objetos (implementan la interfaz *CollisionI*). Así mismo, en <eMU> se establece que únicamente los elementos interactivos puedan poseer lógica ejecutable a través de eventos. Por ello, los elementos interactivos deben implementar la interfaz *EventLogicI* que implica que deban poseer algún tipo de gestión y almacenamiento de eventos. Por otro lado, estos eventos podrán dispararse de manera indirecta cuando el jugador interactúe con un elemento concreto en una habitación. Esta funcionalidad podrá cubrirse implementando la interfaz *TriggerI*, cuyo método disparará la lógica de los eventos, a no ser que estos tengan un comportamiento configurado para dispararse siempre.

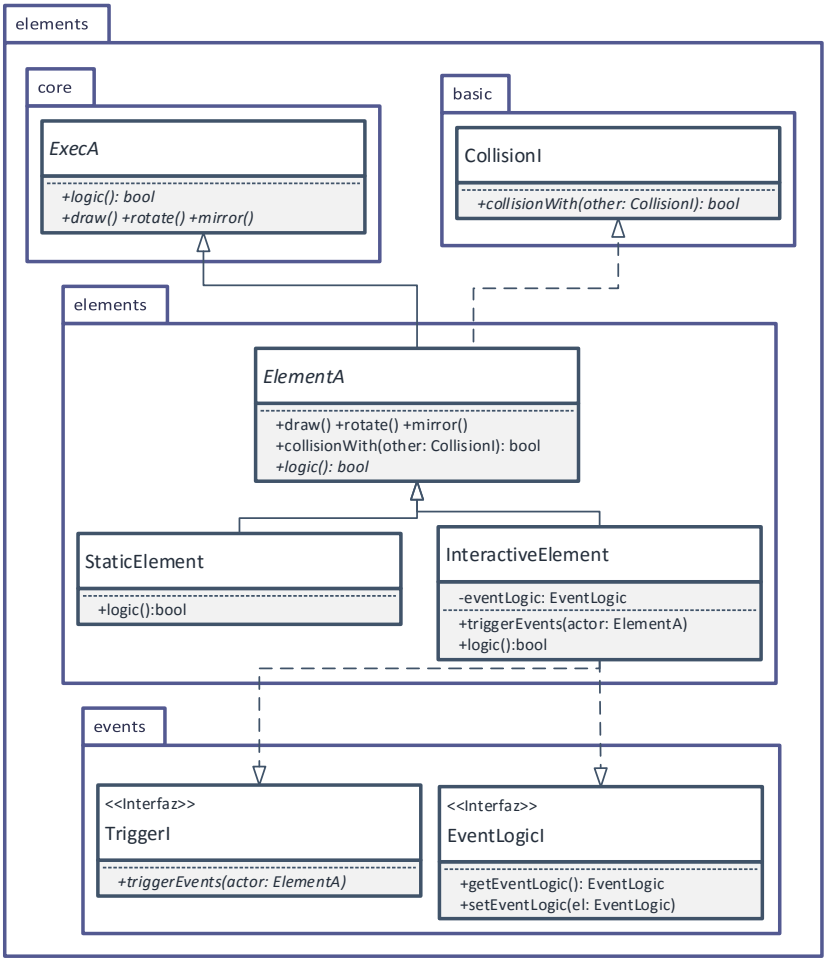


Figura 5.6.6. Modelo de objetos de elementos estáticos e interactivos de <MUe>.

La Figura 5.6.7 muestra las clases e interfaces de las que se componen los eventos. De esta forma, los eventos serán instancias de la clase *EventLogic*. Poseerán, así mismo, una expresión condicional y una lista de instrucciones de tipo *InstructionA*. Si la expresión condicional es evaluada con resultado afirmativo, la lista de instrucciones se ejecutará secuencialmente.

El modelo de objetos de las expresiones condicionales se muestra, por su parte, en la Figura 5.6.8. La evaluación de este tipo de expresión supondrá, por tanto, evaluar la instancia del modelo de objetos de las expresiones condicionales cuya lógica de operandos específica residirá, en último término, en la implementación de las instrucciones condicionales de la clase *ConditionA*.

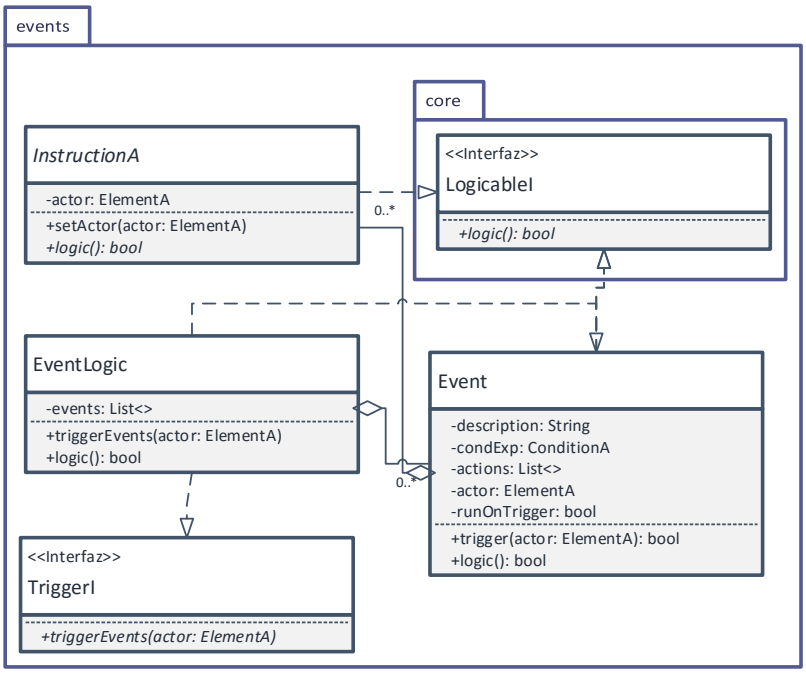


Figura 5.6.7. Modelo de objetos de eventos de <MUe>.

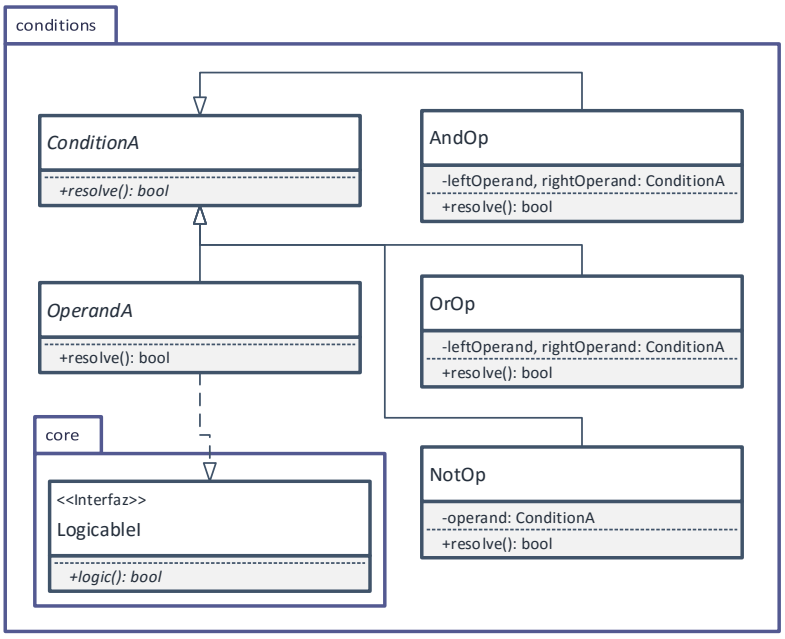


Figura 5.6.8. Modelo de objetos de expresiones condicionales de <MUe>.

5.6.2.2 El marco de juegos <eMU>

eMU\logic\InstructionFactory.java

```
public class InstructionFactory {
  private PlayerLogic playerLogic;  private QuestLogic questLogic; private Inventory inventory;
  private ActiveRoomRenderer activeRoom; private List<Room> rooms;

  public InstructionFactory() {};

  public void setupInstructionFactory(PlayerLogic playerLogic, QuestLogic questLogic,
        Inventory inventory, ActiveRoomRenderer activeRoom, List<Room> rooms) {
    this.playerLogic = playerLogic; this.questLogic = questLogic; this.inventory = inventory; this.activeRoom = activeRoom;
    this.rooms = rooms; }

  public OperandA makeCondInventory(String question, String nameID) {
    switch (question) {
      case "isSelected":
        return new OperandA() {
          @Override
          public boolean logic() {
            if (inventory.getSelectedElement() != null) { return inventory.getSelectedElement().getNameID().equals(nameID); }
            return false;
          }
        };
      case "contains":
        return new OperandA() {
          @Override
          public boolean logic() { return inventory.getElement(nameID) != null; }
        };
    }
  }

  public InstructionA makeInsInventory(String action, int posX, int posY, String nameID) {
    switch (action) {
      case "pick":
        return new InstructionA() {
          @Override
          public boolean logic() {
            ElementA elem = activeRoom.getRender().getElement(nameID);
            if (elem != null) {
              inventory.addToInventory(elem);
              activeRoom.getRender().removeElement(elem);
              return true;
            }
            return false;
          }
        };
      case "place":
        return new InstructionA() {
          @Override
          public boolean logic() {
            ElementA elem = inventory.removeFromInventory(nameID);
            if (elem == null) { return false; }
            elem.setPosXY(posX, posY);
            activeRoom.getRender().addElement(elem, true);
            return true;
          }
        };
    }
  }
...
```

Figura 5.6.9. Porción de la clase factoría de instrucciones condicionales y ejecutables de <eMU>.

El marco de juegos <eMU> acota el tipo de juegos generables mediante <eMU>. Básicamente, dicho marco integra un inventario con funcionalidad para almacenar y extraer objetos, una lógica de jugador que gestiona la realización de las acciones de movimiento e interacción de los jugadores presentes en el escenario, una lógica de misiones que realiza la gestión del estado de las misiones y tareas, y su visualizado (ocultando las tareas y misiones que no son visibles), el ejecutor envoltorio que alberga la habitación actual que se ejecutará y dibujará en todo momento, y la factoría de instrucciones.

La Figura 5.6.9 muestra un fragmento de la factoría de instrucciones. Dicha factoría realizará la construcción, según parámetros de entrada, de las instrucciones condicionales (y expresiones condicionales) y las instrucciones ejecutables de los eventos, mediante la creación de una implementación específica de las clases abstractas *ConditionA e InstructionA* en el motor <MUe>, concretando sus métodos abstractos *logic*.

5.6.3 Desarrollo del generador <eMU> con XLOP3

En esta sección describiremos el desarrollo del generador <eMU> siguiendo la metodología de desarrollo con gramáticas de atributos multivista. La sección 5.6.3.1 plantea el problema de procesamiento a llevar a cabo. La sección 5.6.3.2 presenta la gramática EBNF de base. La sección 5.6.3.3 describe las sintaxis de las distintas vistas. La sección 5.6.3.4 describe la semántica y, por último, la integración de dichas vistas con el marco de aplicación <eMU>.

5.6.3.1 Planteamiento del problema

El generador <eMU> debe procesar documentos que especifiquen juegos <eMU>, instanciar adecuadamente el marco de aplicación subyacente, y ejecutar los juegos así creados. De esta forma, el generador debe abordar dos objetivos básicos:

- El primer objetivo es construir los escenarios en su totalidad. Esta construcción incluye la creación y vinculación de misiones y tareas, habitaciones, elementos y eventos, en función de cómo estos se estructuran en el documento (sección 5.6.1.2), y en función de qué propiedades complementarias (como ubicación en celdas en una habitación) presentan. Estas construcciones se realizarán modelizando los distintos componentes como instancias de objetos del motor de juego <MUe>, a partir de sus diferentes modelos de objetos (sección 5.6.2).
- El segundo objetivo es configurar el juego mediante la creación de distintos componentes adicionales (los cuales permitirán, entre otros aspectos, presentar los elementos visuales, o establecer un sistema de interacción del usuario con la aplicación), para poder producir finalmente un juego operativo.

Para lograr el primer objetivo, será necesario realizar las siguientes tareas de procesamiento:

- Tarea **questCtn**: construcción de misiones. Modeliza las misiones y sus tareas. Para ello, todas las tareas de una misión se almacenan en una lista, manteniendo el mismo orden de aparición en el documento, y posteriormente, se asigna dicha lista de tareas a su misión correspondiente. El orden de las tareas implicará su identificador unívoco de número de tarea, que comenzará en 1, y será relativo a cada misión**.**

- Tarea **roomCtn**: construcción de habitación. Modeliza una habitación y en ella se le asignan los elementos que alberga. Estos elementos se asignarán selectivamente a la instancia de habitación en dos clases de listas:

    - Elementos de *fondo*. Esta será la lista de elementos estáticos. Los elementos estáticos se dibujarán en primer lugar, mostrándose visualmente por debajo de los elementos de primer plano. El orden de los elementos en esta lista no es relevante.

    - Elementos de primer plano. Esta será la lista de elementos interactivos (incluidos jugadores). Estos elementos se dibujarán en segundo lugar, mostrándose visualmente por encima de los elementos de fondo. El orden de los elementos en esta lista no es relevante.

- Tarea **elemCtn**: construcción de un elemento y de otras estructuras básicas. Modeliza un elemento según su clase y, si corresponde, le asigna sus eventos. También modeliza construcciones básicas elementales. Las clases de elementos que construye son:

    - Elementos estáticos. Les asigna un nombre, estados con descripción e información de representación visual y otras propiedades obtenidas del documento.

    - Elementos interactivos. Aparte de la misma información que para los elementos estáticos, les asigna también una lista con los eventos definidos para los mismos.

    - Elementos jugador. Son elementos interactivos con una lógica específica de jugador.

    De esta forma, la asignación de eventos ocurrirá solo para elementos interactivos o elementos jugadores.

- Tarea **eventCtn**: construcción de eventos. Modeliza los eventos de un elemento. Mediante eventos se establece la lógica de transición entre las distintas habitaciones, la gestión del inventario, el cambio de estado del juego, de las misiones, de los elementos, etc. Como ha se ha indicado, un evento consta de una expresión condicional y de una lista de instrucciones (pueden ser vacías). Una expresión condicional se construye también mediante su modelizado, a partir del

modelo de objetos de expresiones condicionales del motor de juego <MUe>, siendo, en última instancia, los componentes de estas expresiones, instancias de una instrucción condicional. Estas instrucciones condicionales se construyen mediante una factoría de instrucciones del marco de juegos <eMU>, que aportan nuevas implementaciones de lógica de ejecución a este tipo de instrucciones. Las instrucciones que construye esta factoría pueden ser condicionales o instrucciones ejecutables. Una instrucción ejecutable puede ser de los siguientes tipos:

- Instrucción de mensaje. Muestra el mensaje especificado (en el documento) al usuario.
- Instrucción de misión. Establece el estado de una tarea de una misión como completa/incompleta o visible/invisible. El identificador clave de referencia es el nombre de la misión y el número de la tarea de la misión relativa a ésta.
- Instrucción de inventario. Añade/elimina un elemento al/del inventario. El identificador clave de referencia es el nombre del elemento. La acción real de añadir o eliminar es mover un elemento desde la habitación donde se encuentra el jugador a su inventario, o colocar un elemento del inventario del jugador en un lugar determinado en la habitación donde se encuentra el jugador. Los elementos no se crean, duplican, ni se eliminan (sección 5.6.1.1).
- Instrucción de movimiento. Cambia la ubicación de un elemento de una habitación origen a una habitación destino, en una posición y dirección local a la habitación destino. El identificador clave de referencia es el nombre del elemento, refiriéndose con “PLAYER” (como palabra reservada) a todos los elementos jugadores a la vez.
- Instrucción de estado. Cambia el estado de un elemento, de una habitación, del inventario del jugador, o del elemento seleccionado en el inventario. El identificador clave de referencia es el nombre del estado.

Por su parte, las instrucciones de condición pueden ser de los siguientes tipos:

- Condición de misión. Comprueba si se cumple el valor de cierto o de falso, en función de formular la pregunta de si la tarea indicada de la misión especificada está visible, o completa.
- Condición de inventario. Comprueba si se cumple el valor de cierto o de falso, en función de formular la pregunta de si el elemento indicado está seleccionado en el inventario, o si el elemento existe en el inventario.
- Condición de estado. Comprueba si se cumple el valor de cierto o de falso, en función de formular la pregunta de si el estado indicado es el actual para

un elemento en una habitación, en el inventario, o aquel que está seleccionado en el inventario.

Para lograr el segundo objetivo, será necesario realizar las siguientes tareas de procesamiento:

- Tarea **setup**: configuración de la aplicación. Realiza la configuración de la aplicación y establece una lógica específica de funcionamiento. Esto implica la construcción y configuración de: un controlador de aplicación, una interfaz de usuario, un registrador de eventos de entrada, un renderizador gráfico, una lógica de movimiento e interacción de jugador y gestión de un inventario, una instancia de modelo de inventario, una lógica de gestión de misiones, y la factoría de instrucciones. Todos estos elementos serán instancias directas del marco de aplicación de <eMU>. Estos elementos requerirán conectarse entre sí como configuración inicial y/o asignarles estructuras modelizadas de las tareas anteriormente expuestas. No obstante, será necesario realizar subtareas adicionales para completar la configuración de todos los componentes:
    - El renderizador gráfico requiere la habitación inicial del juego. Esta habitación será aquella donde figure el jugador número 1. Para ello, se deberá buscar qué habitación contiene dicho jugador, lo que implica visitar todas las habitaciones y, únicamente, inspeccionar los elementos jugadores que existan en ella.
    - La lógica del jugador requiere disponer de todos los jugadores existentes. Para ello, será necesario realizar una búsqueda similar a la anteriormente expuesta, en busca de tales elementos jugador.
    - La factoría de instrucciones requiere disponer de todas las habitaciones existentes, para así poder los jugadores transitar de una habitación a otra, o poder mover elementos entre habitaciones distintas. Para ello, será necesario recopilar una lista con todas las habitaciones existentes en el documento.
- Tarea **prop**: propagación del entorno. Propaga el entorno desde la raíz del árbol de derivación hasta las hojas. Este entorno consistirá en la factoría de instrucciones necesaria para la construcción de las instrucciones condicionales e instrucciones ejecutables, de los eventos.

La designación empleada para nombrar cada una de las tareas anteriores se utilizará para referirse a sus correspondientes vistas sintácticas en la gramática de atributos multivista (sección 5.6.3.3). Mientras tanto, el desarrollo de la capa lógica de la aplicación se puede realizar en paralelo a la elaboración de la gramática de atributos multivista, la cual conforma la capa lingüística de la aplicación.

5.6.3.2 Establecimiento de la gramática EBNF de base

La Figura 5.6.10 muestra la gramática EBNF de base para <eMU> obtenida a partir de su DTD (5.6.1.2). Como puede observarse, en su diseño se han realizado sustituciones directas de no terminales para adecuar la gramática a una estructura más apta y directa de base, sobre la cual modelizar gramáticas BNF enfocadas a las tareas más específicas de procesamiento.

```
Game -> <game> Quests Rooms </game>;
Quests -> Quest+;
Quest -> <quest> NameID Description (<task>#pcdata</task>)+ </quest>;
Rooms -> Room+;
Room -> <room> NameID Description <room.attributes></room.attributes> (Player | Static | Interactive)* </room>;
Description -> (<description>#pcdata</description>)?;
NameID -> <nameID>#pcdata</nameID>;
Interactive -> <interactive> ElementDescription Events </interactive>;
Player -> <player> ElementDescription Events </player>;
Static -> <static> ElementDescription </static>;
ElementDescription -> NameID <elem.attributes></elem.attributes> State+;
State -> <state> NameID Description <state.attributes></state.attributes> Tiles </state>;
Tiles -> (<tiles> (<tile>#pcdata</tile>)* </tiles>)?;
Events -> Event*;
Event -> <event> Description (<conditionExp> c_exp </conditionExp>)? <actions> Instruction+ </actions> </event>;
c_exp -> c_opnd ((<OR></OR>)? c_opnd)*;
c_opnd -> <NOT> c_opnd </NOT> | Condition | <P> c_exp </P>;
Condition -> C_Inventory | C_Quest | C_State;
C_Inventory -> <cond.inventory>#pcdata</cond.inventory>;
C_Quest -> <cond.quest>#pcdata</cond.quest>;
C_State -> <cond.state>#pcdata</cond.state>;
Instruction -> I_Quest | I_Inventory | I_MoveTo | I_Message | I_State;
I_Quest -> <ins.quest>#pcdata</ins.quest>;
I_Inventory -> <ins.inventory>#pcdata</ins.inventory>;
I_MoveTo -> <ins.moveTo>#pcdata</ins.moveTo>;
I_Message -> <ins.message>#pcdata</ins.message>;
I_State -> <ins.state>#pcdata</ins.state>;
```

Figura 5.6.10. Gramática EBNF de base de <eMU> en sintaxis XLOP3.

5.6.3.3 Caracterización de la sintaxis de las vistas

Cada una de las tareas de procesamiento descritas en el planteamiento del problema (sección 5.6.3.1) se definirán con una vista (utilizando el mismo nombre que figura en dicha sección para su tarea asociada). La estructura de cada una de estas vistas se especificará mediante estructuras gramaticales XLOP3. Las Figura 5.6.11, Figura 5.6.12 y Figura 5.6.13, muestran cómo es la estructura de cada una de las vistas asociadas a las tareas de procesamiento **setup**, **roomCtn**, **prop**, **questCtn**, **elemCtn** y **eventCtn**. Como puede observarse, las vistas *questCtn* y *elemCtn* utilizan en su totalidad una estructura reconocedora, mientras que las demás vistas comparten una porción de dicha estructura. En concreto, la vista *setup* comparte la mayoría de esta estructura, excepto la parte asociada con los símbolos de núcleo *Game*, *Rooms* y *Room*. Esto es debido a que necesita una estructura diferente apta para realizar su tarea de procesamiento de configuración inicial del juego, que requiere procesar las

habitaciones y los elementos del documento XML de una manera diferente (realiza búsquedas selectivas). Lo mismo sucede con *Room* para las vistas *roomCtn* y *prop*, en donde la primera necesita recolectar la información de los elementos de una habitación de una manera diferente, y la segunda discriminar según los tipos de los elementos.

<table>
<tr><th>Vista setup</th><th>Vista roomCtn</th><th>Vista prop</th><th>Vista questCtn</th><th>Vista elemCtn</th><th>Vista eventCtn</th></tr>
<tr><td>Structure of Game in setup {
Game ::= Factory <game> Quests Rooms </game> Setup;
Factory ::= ;
Setup ::= KeyListener Controller Render PlayerLogic Inventory QuestLogic InstructionFactory;
KeyListener ::= ;
Controller ::= ;
Render ::= ;
PlayerLogic ::= ;
Inventory ::= ;
QuestLogic ::= ;
InstructionFactory ::= ;
}</td><td colspan="5">Structure of Game in questCtn, roomCtn, elemCtn, eventCtn, prop {
Game ::= <game> Quests Rooms </game>;
}</td></tr>
<tr><td>Structure of Rooms in setup {
Rooms ::= RRooms;
RRooms ::= SRoom RRooms;
RRooms ::= SRoom;
SRoom ::= Room;
}</td><td colspan="5">Structure of Rooms in questCtn, roomCtn, elemCtn, eventCtn, prop {
Rooms ::= RRooms;
RRooms ::= RRooms Room;
RRooms ::= Room;
}</td></tr>
<tr><td colspan="6">Structure of Quests in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
Quests ::= RQuests;
RQuests ::= RQuests Quest;
RQuests ::= Quest;
}</td></tr>
<tr><td colspan="6">Structure of Quest in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
Quest ::= <quest> NameID Description Tasks </quest>;
Tasks ::= Tasks Task;
Tasks ::= Task;
Task ::= <task> #pcdata </task>;
}</td></tr>
<tr><td>Structure of Room in setup {
Room ::= <room> NameID Description <room.attributes> </room.attributes> Elements </room>;
Elements ::= Elements Player;
Elements ::= Elements Other;
Elements ::= ;
Other ::= Interactive;
Other ::= Static;
}</td><td>Structure of Room in roomCtn {
Room ::= <room> NameID Description <room.attributes> </room.attributes> Elements </room>;
Elements ::= Foreground Elements;
Elements ::= Background Elements;
Elements ::= ;
Foreground ::= Player;
Foreground ::= Interactive;
Background ::= Static;
}</td><td>Structure of Room in prop {
Room ::= <room> NameID Description <room.attributes> </room.attributes> Elements </room>;
Elements ::= ElementWithEvents Elements;
Elements ::= Other Elements;
Elements ::= ;
ElementWithEvents ::= Player;
ElementWithEvents ::= Interactive;
Other ::= Static;
}</td><td colspan="3">Structure of Room in elemCtn, questCtn, eventCtn {
Room ::= <room> NameID Description <room.attributes> </room.attributes> Elements </room>;
Elements ::= Elements Element;
Elements ::= ;
Element ::= Player;
Element ::= Static;
Element ::= Interactive;
}</td></tr>
</table>

Figura 5.6.11. Vistas sintácticas de <eMU> para XLOP3 (parte 1).

Vista setup	Vista roomCtn	Vista prop	Vista questCtn	Vista elemCtn	Vista eventCtn

Structure of ElementDescription in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
ElementDescription ::= NameID <elem.attributes> </elem.attributes> States;
States ::= State States;
States ::= State;
}					

Structure of State in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
State ::= <state> NameID Description <state.attributes> </state.attributes> Tiles </state>;
}					

Structure of Static in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
Static ::= <static> ElementDescription </static>;
}					

Structure of Interactive in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
Interactive ::= <interactive> ElementDescription Events </interactive>;
}					

Structure of Player in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
Player ::= <player> ElementDescription Events </player>;
}					

Structure of NameID in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
NameID ::= <nameID> #pcdata </nameID>;
}					

Structure of Description in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
Description ::= <description> #pcdata </description>;
Description ::= ;
}					

Structure of Tiles in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
Tiles ::= <tiles> RTiles </tiles>;
Tiles ::= ;
RTiles ::= RTiles Tile;
RTiles ::= ;
Tile ::= <tile> #pcdata </tile>;
}					

Structure of Events in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
Events ::= REvents;
REvents ::= REvents Event;
REvents ::= ;
}					

Structure of Event in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
Event ::= <event> Description ConditionExp Actions </event>;
ConditionExp ::= <conditionExp> c_exp </conditionExp>;
ConditionExp ::= ;
Actions ::= <actions> Instructions </actions>;
Instructions ::= Instructions Instruction;
Instructions ::= Instruction;
}					

Figura 5.6.12. Vistas sintácticas de <eMU> para XLOP3 (parte 2).

La Figura 5.6.13 muestra la parte estructural que permite procesar expresiones condicionales. La asociatividad y precedencia de estas expresiones sólo queda caracterizada en la vista *eventCtn*, debido a que exclusivamente su tarea realizará la construcción de expresiones condicionales conforme al marco de aplicación de <eMU>. Nótese que la estructura de vistas utilizada para estas expresiones se ha realizado de manera muy similar a la elaborada en la sección 5.2.3.

Vista setup	Vista roomCtn	Vista prop	Vista questCtn	Vista elemCtn	Vista eventCtn

Structure of c_exp in questCtn, roomCtn, elemCtn, prop, setup {
c_exp ::= c_opnd rexp;
rexp ::= op c_opnd rexp;
rexp ::= ;
op ::= <OR> </OR>;
op ::= ;
}
					Structure of c_exp in eventCtn {
					c_exp ::= expr;
					expr ::= expr <OR> </OR> term;
					expr ::= term;
					term ::= term c_opnd;
					term ::= c_opnd;
					}

Structure of c_opnd in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
c_opnd ::= <NOT> c_opnd </NOT>;
c_opnd ::= Condition;
c_opnd ::= <P> c_exp </P>;
}					

Structure of Condition in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
Condition ::= C_Quest;
Condition ::= C_Inventory;
Condition ::= C_State;
}					

Structure of C_Quest in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
C_Quest ::= <cond.quest> #pcdata </cond.quest>;
}					

Structure of C_Inventory in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
C_Inventory ::= <cond.inventory> #pcdata </cond.inventory>;
}					

Structure of C_State in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
C_State ::= <cond.state> #pcdata </cond.state>;
}					

Structure of Instruction in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
Instruction ::= I_Quest;
Instruction ::= I_Inventory;
Instruction ::= I_Message;
Instruction ::= I_MoveTo;
Instruction ::= I_State;
}					

Structure of I_Quest in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
I_Quest ::= <ins.quest> #pcdata </ins.quest>;
}					

Structure of I_Inventory in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
I_Inventory ::= <ins.inventory> #pcdata </ins.inventory>;
}					

Structure of I_Message in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
I_Message ::= <ins.message> #pcdata </ins.message>;
}					

Structure of I_MoveTo in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
I_MoveTo ::= <ins.moveTo> #pcdata </ins.moveTo>;
}					

Structure of I_State in questCtn, roomCtn, elemCtn, eventCtn, prop, setup {
I_State ::= <ins.state> #pcdata </ins.state>;
}					

Figura 5.6.13. Vistas sintácticas de <eMU> para XLOP3 (parte 3).

En la siguiente sección se describirá la semántica de cada una de las vistas. Para ello, a las estructuras gramaticales introducidas anteriormente se le asociarán, si procede y según vista, una especificación de semántica gramatical XLOP3.

5.6.3.4 Caracterización de la semántica de las vistas

Establecida la estructura de cada una de las vistas *setup*, *roomCtn*, *prop*, *questCtn*, *elemCtn* y *eventCtn* con estructuras gramaticales XLOP3, se procede a caracterizar cada una de sus tareas de procesamiento aportando semánticas gramaticales XLOP3. A continuación, se presenta cada una de las semánticas gramaticales para las vistas mediante figuras, en donde aparecen atributos de símbolos de núcleo subrayados para indicar que sus valores son sintetizados y procedentes de vistas externas, lo contrario a aquellos que aparecen resaltados y que indican el establecimiento del valor de los atributos heredados para estar disponibles y que serán utilizados en vistas externas.

La semántica de la vista *setup* (Figura 5.6.14) se constituye como sigue:

- La configuración de <eMU> requiere la creación y configuración de una serie de componentes, que se realizan principalmente en la semántica gramatical para *Game*. El componente de lógica de gestión de misiones necesita la lista de misiones del juego, la cual se obtiene a través del atributo *list* de la vista *questCtn* de *Quests* a través de la producción *Game*. En dicha producción se inyecta a *Rooms* la factoría de nodos, a través de su atributo heredado *ifach*, ya que será necesaria en alguna vista externa.

- La factoría de nodos requiere una lista con todas las habitaciones del juego. Esta lista se construye en la semántica gramatical para *Rooms*, en donde cada habitación se obtiene del atributo sintetizado *r* de *Room* procedente de la vista que la construye (*roomCtn*).

- El renderizador gráfico requiere la habitación inicial del juego (aquella donde aparece el jugador nº 1), por lo que se realiza una exploración selectiva en las habitaciones y de los elementos, en las semánticas gramaticales *Rooms* y *Room*, utilizándose los atributos *sr* e *isSel*.

- La lógica del jugador se construye y se propaga por *Rooms* y *Room* con *plh*, añadiendo los jugadores a este componente. Para que esto sea efectivo, es necesario propagar sintentizadamente el componente con *pl* y solicitarlo en la función inicializadora del juego (función semántica *run* asociada al atributo sintetizado *run* de *Game*, nodo raíz de vista).

Vista setup

```
Semantics for Game of setup {
  Game ::= Factory  <game> Quests Rooms </game> Setup {
    run of Game = run(c of Setup, setRenderRoom(sr of Rooms, r of Setup), pl of Rooms, inv of Setup, ql of Setup, ifac of Setup);
    plh of Rooms = pl of Setup;     ifach of Rooms = f of Factory;
    roomsh of Setup = list of Rooms;     ifach of Setup = f of Factory;     qsh of Setup = list[questCtn] of Quests;
  }
  Factory ::= {     f of Factory = newInstructionFactory();   }
  Setup ::= KeyListener Controller Render PlayerLogic Inventory QuestLogic InstructionFactory {
    klh of Controller = kl of KeyListener;
    rh of PlayerLogic = r of Render;     invh of PlayerLogic = inv of Inventory;     klh of PlayerLogic = kl of KeyListener;
    qsh of QuestLogic = qsh of Setup;     ch of QuestLogic = c of Controller;
    plh of InstructionFactory = pl of PlayerLogic;     qlh of InstructionFactory = ql of QuestLogic;
    ch of Inventory = c of Controller;
    invh of InstructionFactory = inv of Inventory;     rh of InstructionFactory = r of Render;     roomsh of InstructionFactory = roomsh of Setup;
    ifach of InstructionFactory = ifach of Setup;
    c of Setup = addGUI(c of Controller);     r of Setup = r of Render;     pl of Setup = pl of PlayerLogic;
    inv of Setup = inv of Inventory;     ql of Setup = ql of QuestLogic;     ifac of Setup = ifac of InstructionFactory;
  }
  KeyListener ::= {     kl of KeyListener = newKeyListener();   }
  Controller ::= {     c of Controller = newController(klh of Controller);   }
  Render ::= {     r of Render = newRender();   }
  PlayerLogic ::= {     pl of PlayerLogic = newPlayerLogic(rh of PlayerLogic, invh of PlayerLogic, klh of PlayerLogic);   }
  Inventory ::= {     inv of Inventory = newInventory(ch of Inventory);   }
  QuestLogic ::= {     ql of QuestLogic = newQuestLogic(qsh of QuestLogic, ch of QuestLogic);   }
  InstructionFactory ::= {
    ifac of InstructionFactory = setupInstructionFactory(plh of InstructionFactory, qlh of InstructionFactory,
    invh of InstructionFactory, rh of InstructionFactory, roomsh of InstructionFactory, ifach of InstructionFactory);
  }
}

Semantics for Rooms of setup {
  Rooms ::= RRooms {
    list of Rooms = list of RRooms;     sr of Rooms = sr of RRooms;     pl of Rooms = pl of RRooms;
    plh of RRooms = plh of Rooms;
  }
  RRooms ::= SRoom RRooms{
    list of RRooms = addList(r of SRoom, list of RRooms(1));     sr of RRooms = selectNotNull(sr of RRooms(1), sr of SRoom);
    plh of RRooms(1) = pl of SRoom;
    plh of SRoom = plh of RRooms;
    pl of RRooms = pl of RRooms(1);
  }
  RRooms ::= SRoom {
    list of RRooms = addList(r of SRoom, newList());     sr of RRooms = sr of SRoom;     pl of RRooms = pl of SRoom;
    plh of SRoom = plh of RRooms;
  }
  SRoom ::= Room {
    r of SRoom = r[roomCtn] of Room;     sr of SRoom = selectRoom(isSel of Room, r[roomCtn] of Room);     pl of SRoom = pl of Room;
    plh of Room = plh of SRoom;
  }
}

Semantics for Room of setup {
  Room ::= <room> NameID Description <room.attributes> </room.attributes> Elements </room> {
    plh of Elements = plh of Room;
    pl of Room = pl of Elements;     isSel of Room = isSel of Elements;
  }
  Elements ::= Elements Player {
    plh of Elements(1) = plh of Elements;
    pl of Elements = addPlayer(e[elemCtn] of Player, id[elemCtn] of Player, pl of Elements(1));
    isSel of Elements = operationOR(hasPlayerOne(id[elemCtn] of Player), isSel of Elements(1));
  }
  Elements ::= Elements Other {
    plh of Elements(1) = plh of Elements;
    pl of Elements = pl of Elements(1);     isSel of Elements = operationOR(toBoolean("false"), isSel of Elements(1));
  }
  Elements ::= {
    pl of Elements = plh of Elements;     isSel of Elements = toBoolean("false");
  }
  Other ::= Interactive {}
  Other ::= Static {}
}
```

Figura 5.6.14. Semánticas de la vista setup de <eMU> para XLOP3.

Por su parte, y en lo que se refiere a la vista *roomCtn* (Figura 5.6.15), con la semántica gramatical para *Room* se construye una habitación, discriminando los elementos de fondo (bajo *Background*) de los elementos de primer plano (bajo *Foreground*), construyendo ambas listas de elementos independientemente. El elemento en sí se obtiene de la vista *elemCtn* a través del atributo *e* dependiendo del tipo de elemento *Player*, *Interactive* o *Static*. También se obtienen de dicha vista otros valores sintetizados necesarios (*v* de *NameID* y de *Description*).

Vista roomCtn

```
Semantics for Room of roomCtn {
  Room ::= <room> NameID Description <room.attributes> </room.attributes> Elements </room> {
    bhlist of Elements = newList();     fhlist of Elements = newList();
    r of Room = addBackground(blist of Elements, addForeground(flist of Elements,
      newRoom(v[elemCtn] of NameID, v[elemCtn] of Description, posX of <room.attributes>, posY of <room.attributes>,
      width of <room.attributes>, height of <room.attributes>, backgroundImage of <room.attributes>)));
  }
  Elements ::= {     blist of Elements = bhlist of Elements;     flist of Elements = fhlist of Elements;   }
  Elements ::= Foreground Elements {
    blist of Elements = blist of Elements(1);     flist of Elements = addList(v of Foreground, flist of Elements(1));
    bhlist of Elements(1) = bhlist of Elements;     fhlist of Elements(1) = fhlist of Elements;
  }
  Elements ::= Background Elements {
    blist of Elements = addList(v of Background, blist of Elements(1));     flist of Elements = flist of Elements(1);
    bhlist of Elements(1) = bhlist of Elements;     fhlist of Elements(1) = fhlist of Elements;
  }
  Foreground ::= Player {     v of Foreground = e[elemCtn] of Player;   }
  Foreground ::= Interactive {     v of Foreground = e[elemCtn] of Interactive;   }
  Background ::= Static {     v of Background = e[elemCtn] of Static;   }
}
```

Figura 5.6.15. Semántica de la vista roomCtn de <eMU> para XLOP3.

La Figura 5.6.16 aborda los aspectos semánticos de la vista *questCtn*. La lista de misiones se construye con la semántica gramatical para *Quests*, y cada misión con la de *Quest*, en donde se obtienen algunos valores sintetizados necesarios de la vista *elemCtn* (*v* de *NameID* y de *Description*).

Vista questCtn

```
Semantics for Quests of questCtn {
  Quests ::= RQuests {     list of Quests = list of RQuests;   }
  RQuests ::= RQuests Quest {     list of RQuests = addList(q of Quest, list of RQuests(1));   }
  RQuests ::= Quest {     list of RQuests = addList(q of Quest, newList());   }
}

Semantics for Quest of questCtn {
  Quest ::= <quest> NameID Description Tasks </quest> {
    q of Quest = newQuest(v[elemCtn] of NameID, v[elemCtn] of Description, list of Tasks);
  }
  Tasks ::= Tasks Task {     list of Tasks = addList(t of Task, list of Tasks(1));   }
  Tasks ::= Task {     list of Tasks = addList(t of Task, newList());   }
  Task ::= <task> #pcdata </task> {     t of Task = newTask(text of #pcdata);   }
}
```

Figura 5.6.16. Semánticas de la vista questCtn de <eMU> para XLOP3.

En lo que se refiere a la vista *prop* (Figura 5.6.17), comenzando desde el símbolo de núcleo *Rooms*, se obtiene de su atributo heredado *ifach*, procedente de la vista *setup*, la factoría de nodos. Esta factoría se propaga también mediante *ifach* por las habitaciones y exclusivamente por los elementos interactivos, como bien se describe en la semántica gramatical para *Room*, pues es requerida en las instrucciones de tipo condicional de *Condition* y de tipo ejecutable de *Instruction* de los eventos *Events* de estos elementos.

Vista prop

```
Semantics for Rooms of prop {
  Rooms ::= RRooms {    ifach of RRooms = ifach[setup] of Rooms;  }
  RRooms ::= RRooms Room {    ifach of RRooms(1) = ifach of RRooms;    ifach of Room = ifach of RRooms;  }
  RRooms ::= Room {    ifach of Room = ifach of RRooms;  }
}

Semantics for Room of prop {
  Room ::= <room> NameID Description <room.attributes> </room.attributes> Elements </room> {    ifach of Elements = ifach of Room;  }
  Elements ::= ElementWithEvents Elements {    ifach of ElementWithEvents = ifach of Elements;    ifach of Elements(1) = ifach of Elements;  }
  Elements ::= Other Elements {    ifach of Elements(1) = ifach of Elements;  }
  Elements ::= {}
  ElementWithEvents ::= Player {    ifach of Player = ifach of ElementWithEvents;  }
  ElementWithEvents ::= Interactive {    ifach of Interactive = ifach of ElementWithEvents;  }
  Other ::= Static {}
}

Semantics for Interactive of prop {
  Interactive ::= <interactive> ElementDescription Events </interactive> {    ifach of Events = ifach of Interactive;  }
}

Semantics for Player of prop {
  Player ::= <player> ElementDescription Events </player> {    ifach of Events = ifach of Player;  }
}

Semantics for Events of prop {
  Events ::= REvents {    ifach of REvents = ifach of Events;  }
  REvents ::= REvents Event {    ifach of REvents(1) = ifach of REvents;    ifach of Event = ifach of REvents;  }
  REvents ::= {}
}

Semantics for Event of prop {
  Event ::= <event> Description ConditionExp Actions </event> {    ifach of ConditionExp = ifach of Event;    ifach of Actions = ifach of Event;  }
  ConditionExp ::= <conditionExp> c_exp </conditionExp> {    ifach of c_exp = ifach of ConditionExp;  }
  ConditionExp ::= {}
  Actions ::= <actions> Instructions </actions> {    ifach of Instructions = ifach of Actions;  }
  Instructions ::= Instructions Instruction {    ifach of Instructions(1) = ifach of Instructions;    ifach of Instruction = ifach of Instructions;  }
  Instructions ::= Instruction {    ifach of Instruction = ifach of Instructions;  }
}

Semantics for c_exp of prop {
  c_exp ::= c_opnd rexp {    ifach of c_opnd = ifach of c_exp;    ifach of rexp = ifach of c_exp;  }
  rexp ::= op c_opnd rexp {    ifach of c_opnd = ifach of rexp;    ifach of rexp(1) = ifach of rexp;  }
  rexp ::= {}
  op ::= <OR> </OR> {}
  op ::= {}
}

Semantics for c_opnd of prop {
  c_opnd ::= <NOT> c_opnd </NOT> {    ifach of c_opnd(1) = ifach of c_opnd;  }
  c_opnd ::= Condition {    ifach of Condition = ifach of c_opnd;  }
  c_opnd ::= <P> c_exp </P> {    ifach of c_exp = ifach of c_opnd;  }
}

Semantics for Condition of prop {
  Condition ::= C_Quest {    ifach of C_Quest = ifach of Condition;  }
  Condition ::= C_Inventory {    ifach of C_Inventory = ifach of Condition;  }
  Condition ::= C_State {    ifach of C_State = ifach of Condition;  }
}

Semantics for Instruction of prop {
  Instruction ::= I_Quest {    ifach of I_Quest = ifach of Instruction;  }
  Instruction ::= I_Inventory {    ifach of I_Inventory = ifach of Instruction;  }
  Instruction ::= I_Message {    ifach of I_Message = ifach of Instruction;  }
  Instruction ::= I_MoveTo {    ifach of I_MoveTo = ifach of Instruction;  }
  Instruction ::= I_State {    ifach of I_State = ifach of Instruction;  }
}
```

Figura 5.6.17. Semánticas de la vista prop de <eMU> para XLOP3.

La vista *elemCtn* se trata en la Figura 5.6.18. Respecto a ella, las semánticas gramaticales *Player*, *Interactive* y *Static* establecen el tipo de elemento a construir, el cual es creado en *ElementDescription* y, posteriormente, completado si lo requiere, con una lógica de eventos construida a partir de la lista de eventos del atributo sintetizado *list* de *eventCtn* de *Events*. Las restantes semánticas gramaticales permiten aportar la información necesaria para la construcción de estos elementos, y a su vez, funcionar como tareas de obtención de valores en *NameID y Description* que son empleados en otras vistas.

Vista elemCtn

```
Semantics for Player of elemCtn { //Builds an Element of a type.
  Player ::= <player> ElementDescription Events </player> {
    id of Player = id of <player>;     e of Player = addEventLogicToElement(newEventLogic(list[eventCtn] of Events), v of ElementDescription);
    typeh of ElementDescription = "player";
  }
}

Semantics for Static of elemCtn {
  Static ::= <static> ElementDescription </static> {
    typeh of ElementDescription = "static";
    e of Static = v of ElementDescription;
  }
}

Semantics for Interactive of elemCtn {
  Interactive ::= <interactive> ElementDescription Events </interactive> {
    typeh of ElementDescription = "interactive";
    e of Interactive = addEventLogicToElement(newEventLogic(list[eventCtn] of Events), v of ElementDescription);
  }
}

Semantics for ElementDescription of elemCtn {
 ElementDescription ::= NameID <elem.attributes> </elem.attributes> States {
    v of ElementDescription = newElement(typeh of ElementDescription, v of NameID,
    posX of <elem.attributes>, posY of <elem.attributes>, lookTo of <elem.attributes>, initState of <elem.attributes>, list of States);
 }
  States ::= State States {     list of States = addList(v of State, list of States(1));  }
  States ::= State {     list of States = addList(v of State, newList());  }
}

Semantics for State of elemCtn {
  State ::= <state> NameID Description  <state.attributes> </state.attributes> Tiles </state> {
    v of State = newState(v of NameID, v of Description, passable of <state.attributes>, hidden of <state.attributes>, newCompositeBlock(list of
Tiles));
  }
}

Semantics for NameID of elemCtn {
  NameID ::= <nameID> #pcdata </nameID> {     v of NameID = text of #pcdata;  }
}

Semantics for Description of elemCtn {
  Description ::= <description> #pcdata </description> {     v of Description = text of #pcdata;  }
  Description ::= {     v of Description = "";  }
}

Semantics for Tiles of elemCtn {
  Tiles ::= <tiles> RTiles </tiles> {     list of Tiles = list of RTiles;  }
  Tiles ::= {     list of Tiles = newList();  }
  RTiles ::= RTiles Tile {     list of RTiles = addList(v of Tile, list of RTiles(1));  }
  RTiles ::= {     list of RTiles = newList();  }
  Tile ::= <tile> #pcdata </tile> {     v of Tile = newBlock(posX of <tile>, posY of <tile>, mirror of <tile>, lookTo of <tile>, text of #pcdata);  }
}
```

Figura 5.6.18. Semántica de la vista elemCtn de <eMU> para XLOP3.

La semántica de la vista *eventCtn* se plasma en las Figura 5.6.19 y Figura 5.6.20. La semántica gramatical para *Events* construye la lista de eventos. Cada evento de *Event* se construye con una expresión condicional y una lista de instrucciones ejecutables. Las expresiones condicionales requieren la factoría de instrucciones para ser creadas, la cual se obtiene del atributo heredado *ifach* de la vista *prop* de *Event*. De la misma manera, cada una de las instrucciones tanto condicionales como ejecutables, requieren el mismo componente, el cual se obtiene también a través de ese atributo heredado de la vista *prop*, pero respecto al símbolo de núcleo correspondiente de la instrucción.

Vista eventCtn

```
Semantics for Events of eventCtn {
  Events ::= REvents {     list of Events = list of REvents;   }
  REvents ::= REvents Event {     list of REvents = addList(e of Event, list of REvents(1));   }
  REvents ::= {     list of REvents = newList();   }
}

Semantics for Event of eventCtn  {
  Event ::= <event> Description ConditionExp Actions </event> {
    e of Event = newEvent(v[elemCtn] of Description, trigger of <event>, c of ConditionExp, a of Actions);
    ifach of ConditionExp = ifach[prop] of Event;   }
  ConditionExp ::= <conditionExp> c_exp </conditionExp> {     c of ConditionExp = c of c_exp;   }
  ConditionExp ::= {     c of ConditionExp = newCondition("true", ifach of ConditionExp);   }
  Actions ::= <actions> Instructions </actions> {     a of Actions = list of Instructions;   }
  Instructions ::= Instructions Instruction {     list of Instructions = addList(i of Instruction, list of Instructions(1));   }
  Instructions ::= Instruction {     list of Instructions = addList(i of Instruction, newList());   }
}

Semantics for c_exp of eventCtn {
  c_exp  ::= expr {     c of c_exp = c of expr;   }
  expr ::= expr <OR> </OR> term {     c of expr = newOrCondition(c of expr(1), c of term);   }
  expr ::= term {     c of expr = c of term;   }
  term ::= term c_opnd {     c of term = newAndCondition(c of term(1), c of c_opnd);   }
  term ::= c_opnd {     c of term = c of c_opnd;   }
}

Semantics for c_opnd of eventCtn {
  c_opnd ::= <NOT> c_opnd </NOT> {     c of c_opnd = newNotCondition(c of c_opnd(1));   }
  c_opnd ::= Condition {     c of c_opnd = c of Condition;   }
  c_opnd ::= <P> c_exp </P> {     c of c_opnd = c of c_exp;   }
}

Semantics for Condition of eventCtn {
  Condition ::= C_Quest {     c of Condition = c of C_Quest;   }
  Condition ::= C_Inventory {     c of Condition = c of C_Inventory;   }
  Condition ::= C_State {     c of Condition = c of C_State;   }
}

Semantics for C_Quest of eventCtn {
  C_Quest ::= <cond.quest> #pcdata </cond.quest> {
    c of C_Quest = newConditionQuest(question of <cond.quest>, taskNumber of <cond.quest>, text of #pcdata, ifach[prop] of C_Quest);
  }
}

Semantics for C_Inventory of eventCtn {
  C_Inventory ::= <cond.inventory> #pcdata </cond.inventory> {
    c of C_Inventory = newConditionInventory(question of <cond.inventory>, text of #pcdata, ifach[prop] of C_Inventory);
  }
}

Semantics for C_State of eventCtn {
  C_State ::= <cond.state> #pcdata </cond.state> {
    c of C_State = newConditionState(question of <cond.state>, actor of <cond.state>, text of #pcdata, ifach[prop] of C_State);
  }
}
```

Figura 5.6.19. Semántica de la vista eventCtn de <eMU> para XLOP3 (parte 1).

Vista eventCtn

```
Semantics for Instruction of eventCtn {
  Instruction ::= I_Quest {     i of Instruction = i of I_Quest;   }
  Instruction ::= I_Inventory {     i of Instruction = i of I_Inventory;   }
  Instruction ::= I_Message {     i of Instruction = i of I_Message;   }
  Instruction ::= I_MoveTo {     i of Instruction = i of I_MoveTo;   }
  Instruction ::= I_State {     i of Instruction = i of I_State;   }
}

Semantics for I_Quest of eventCtn {
  I_Quest ::= <ins.quest> #pcdata </ins.quest> {
     i of I_Quest = newInstructionQuest(action of <ins.quest>, taskNumber of <ins.quest>, text of #pcdata, ifach[prop] of I_Quest);   }
}

Semantics for I_Inventory of eventCtn {
  I_Inventory ::= <ins.inventory> #pcdata </ins.inventory> {
     i of I_Inventory = newInstructionInventory(action of <ins.inventory>, posX of <ins.inventory>, posY of <ins.inventory>, text of #pcdata,
ifach[prop] of I_Inventory);   }
}

Semantics for I_Message of eventCtn {
  I_Message ::= <ins.message> #pcdata </ins.message> {    i of I_Message = newInstructionMessage(text of #pcdata, ifach[prop] of I_Message);   }
}

Semantics for I_MoveTo of eventCtn {
  I_MoveTo ::= <ins.moveTo> #pcdata </ins.moveTo> {
     i of I_MoveTo = newInstructionMoveTo(actor of <ins.moveTo>, lookTo of <ins.moveTo>, posX of <ins.moveTo>, posY of <ins.moveTo>, text of
#pcdata, ifach[prop] of I_MoveTo);   }
}

Semantics for I_State of eventCtn {
  I_State ::= <ins.state> #pcdata </ins.state> {
     i of I_State = newInstructionState(action of <ins.state>, actor of <ins.state>, text of #pcdata, ifach[prop] of I_State);
  }
}
```

Figura 5.6.20. Semántica de la vista eventCtn de <eMU> para XLOP3 (parte 2).

SemanticClass.java

```
public Object newEvent(Object description, Object trigger, Object condition, Object actions) {
  return new Event((String)description, (String)trigger, (ConditionA)condition, (Iterable<InstructionA>) actions);
}
public Object newOrCondition(Object leftCondition, Object rightCondition) {
  return new OrOp((ConditionA)leftCondition,(ConditionA)rightCondition);
}
public Object newAndCondition(Object leftCondition, Object rightCondition) {
  return new AndOp((ConditionA)leftCondition,(ConditionA)rightCondition);
}
public Object newNotCondition(Object condition) {
  return new NotOp((ConditionA)condition);
}
public Object newCondition(Object value, Object insFactory) {
  return ((InstructionFactory)insFactory).makeCondition(Boolean.valueOf((String)value));
}
public Object newConditionQuest(Object question, Object taskNumber, Object questID, Object insFactory) {
  return ((InstructionFactory)insFactory).makeCondQuest((String)question, Integer.valueOf((String)taskNumber), (String)questID);
}
public Object newConditionInventory(Object question, Object questID, Object insFactory) {
  return ((InstructionFactory)insFactory).makeCondInventory((String)question, (String)questID);
}
public Object newConditionState(Object question, Object actor, Object stateID, Object insFactory) {
  return ((InstructionFactory)insFactory).makeCondState((String)question, (String)actor,(String)stateID);
}
```

Figura 5.6.21. Extracto de la clase semántica de <eMU> (creación de eventos e instrucciones condicionales).

Por último, las funciones semánticas a las que se hacen referencia en la semántica de las vistas, la mayoría de carácter constructora de modelos del marco de aplicación de <eMU>, se implementan en la clase semántica. En la Figura 5.6.21 se muestra un extracto de la clase semántica para <eMU> con las funciones semánticas de la vista *eventCtn*.

5.6.4 Resultado

Con la aportación de la lógica específica de la sección 5.6.2 y la capa lingüística de la sección 5.6.3, la generación de la aplicación de procesamiento XML <eMU> se realiza de manera sistemática y automática con XLOP3 de la manera explicada en la sección 5.5.3. Así mismo, se procede con su ejecución, según las indicaciones expuestas en la sección 5.5.4. El resultado que se obtiene es la funcionalidad descrita en la sección 5.6.1.1.

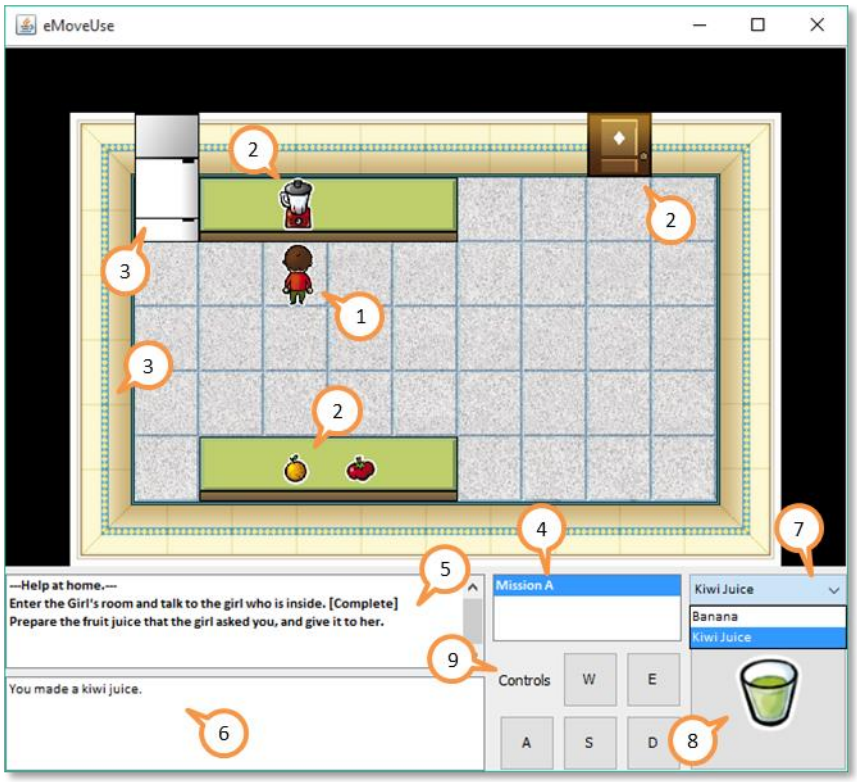


Figura 5.6.22. Interfaz y ejemplo de ejecución de <eMU>.

Obsérvese el ejemplo de ejecución de la Figura 5.6.22. En ella se pueden resaltar las distintas características y funcionalidades de un juego de <eMU>:

1. Elemento jugador. En la figura se muestra mirando hacia arriba porque su estado actual es “Look up”. Se mueve por el escenario con las teclas W, A, S, D., indicadas en 9.

2. Elemento interactivo. Si el usuario interactúa, con la tecla E indicada en 9 mirándolo y estando en frente del elemento, se ejecutan los eventos que contiene. Por ejemplo, la

naranja (*Orange*) pasaría al inventario del jugador si éste interactúa con ella, o la exprimidora (*Blender*) cambiaría el estado de una fruta seleccionada en el inventario al estado zumo (*Juice*).

3. Elemento estático. No puede interactuarse con ellos, pero pueden bloquear el paso al elemento jugador.

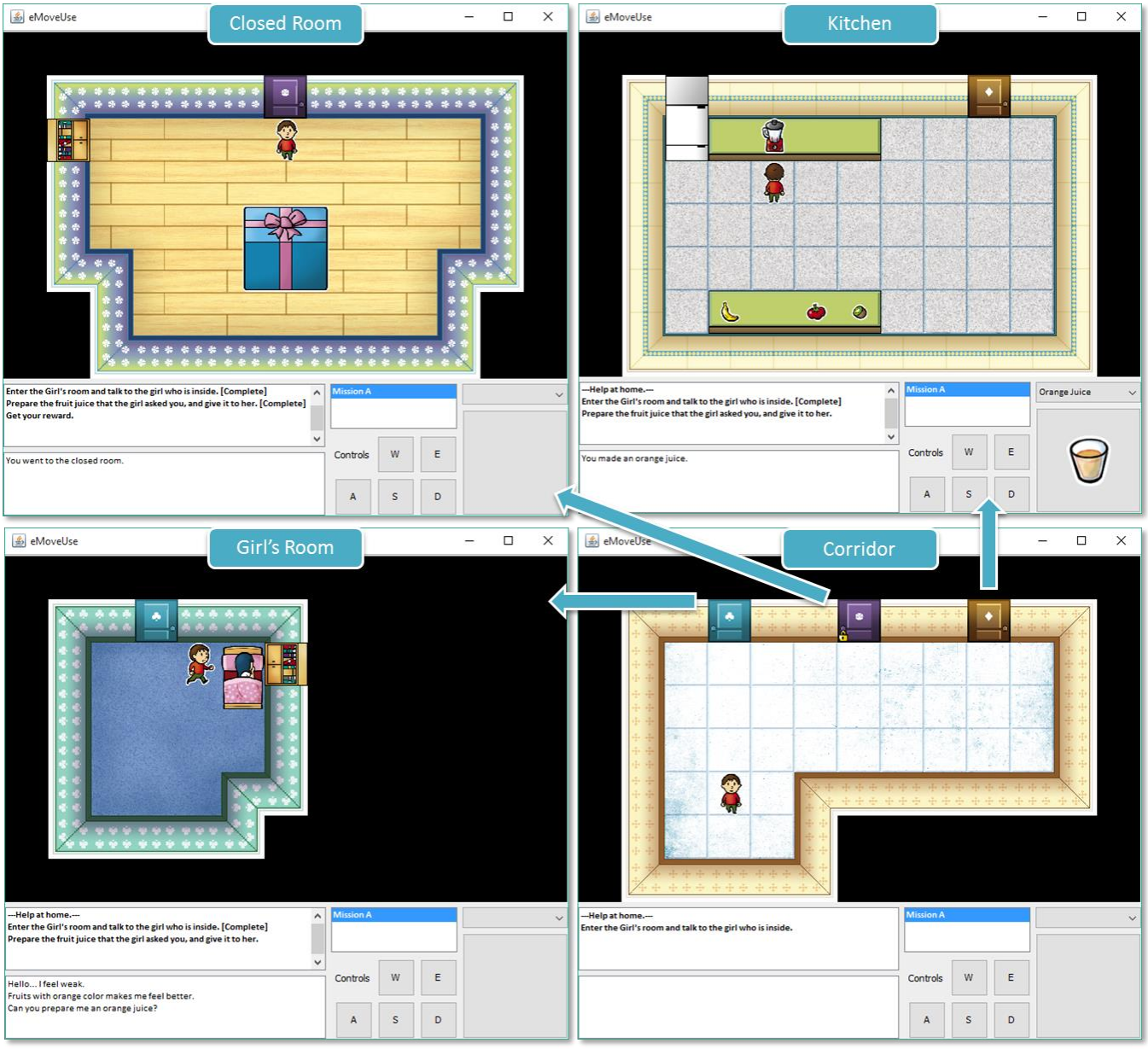


Figura 5.6.23. Resumen del juego generado por el documento XML de *Fruits and Colors* en <eMU>.

4. Muestra las misiones. Permite seleccionar una misión.

5. Muestra las tareas de la misión seleccionada.

6. Cuadro de diálogo que muestra los mensajes producidos por eventos, o descripciones de elementos que mira directamente el elemento jugador.

7. Inventario. Permite seleccionar elementos en el inventario del menú desplegable.

8. Elemento seleccionado en el inventario. Haciendo clic en él se muestra la descripción de dicho elemento en el cuadro de diálogo de 6.

9. Controles accionables mediante clic del ratón.

El documento XML de *Fruits and Colors* de la sección 5.6.1.2, procesado por el generador <eMU> construido con XLOP3, resulta en el juego interactivo educativo con las habitaciones, elementos, misiones e interacciones diseñados en la sección 5.6.1.1, que se resume en la Figura 5.6.23. La resolución de la única misión que presenta (*Mission A*) se realiza de la siguiente manera:

1. El jugador comienza en la habitación *Corridor*, y debe interactuar con el elemento *Door* de la izquierda para ir a la habitación *Girl's Room*.

2. En la habitación *Girl's Room*, el jugador interactúa con el elemento *Girl*, completándose la primera tarea de la misión. Ahora debe volver al pasillo y en él, interactuar con el elemento *Door* de la derecha para ir a la habitación *Kitchen*.

3. En la habitación Kitchen, el jugador interactúa con el elemento *Orange* y éste pasa a su inventario. Posteriormente, el jugador selecciona en su inventario dicho elemento e interactúa con el elemento *Blender*, cambiando el estado del elemento *Orange* a *Orange Juice*. El jugador debe volver a la habitación *Girl's Room*.

4. En la habitación *Girl's Room*, el jugador interactúa con el elemento *Girl*. La segunda tarea de la misión se completa y en el inventario se le añade el elemento *Key*. Ahora el jugador debe volver a la habitación *Corridor*.

5. En la habitación *Corridor*, con el elemento *Key* seleccionado en el inventario, el jugador interactúa con el elemento *Door* en estado *Locked* del centro, y éste cambia de estado, desapareciendo visualmente el candado que presentaba. Al interactuar ahora el jugador con este elemento irá a la habitación *Closed Room*.

6. En la habitación *Closed Room*, el jugador interactúa con el elemento *Present*, y éste desaparece (es ocultado al usuario) y en su lugar aparece el elemento *Girl* con estado diferente. Al interactuar con dicho elemento, se completa la última tarea de la misión, y con ella, la misión en sí, y el juego, presentándose una adicional habitación como pantalla de fin (Figura 5.6.24).

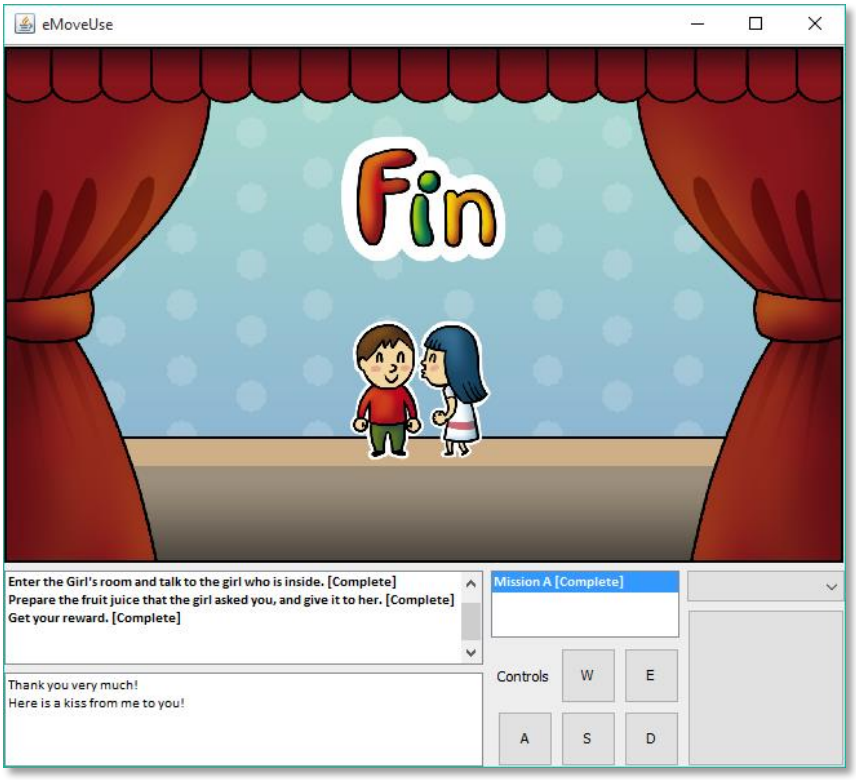


Figura 5.6.24. Finalización del juego *Fruits and Colors* en <eMU>.

5.7 A modo de conclusión

En este capítulo se ha abordado la segunda de las limitaciones detectadas en relación con los enfoques dirigidos por lenguajes al desarrollo de aplicaciones de procesamiento XML formulados en el grupo ILSA: el uso de una gramática común para llevar a cabo las distintas tareas que integran una aplicación de procesamiento XML compleja. Para ello, hemos propuesto el nuevo modelo de las *gramáticas de atributos multivista*, modelo que permite organizar una especificación compleja en vistas gramaticales, cada una de las cuáles puede formularse sobre su propia gramática incontextual. Todas estas gramáticas incontextuales, por otra parte, deben ser conformes con una gramática EBNF de base para el lenguaje de marcado objeto de procesamiento. Por tanto, todas ellas compartirán los mismos símbolos de núcleo, lo que, a su vez, permitirá unificar en un bosque de análisis los distintos árboles de análisis sintáctico producidos por las distintas gramáticas durante el análisis de un documento. La evaluación semántica se llevará a cabo, de esta forma, sobre el citado bosque de análisis.

De esta forma, las gramáticas de atributos multivista comparten los objetivos de las propuestas de modularización de gramáticas de atributos a las que se ha hecho referencia en el Capítulo 2, ya que se centran en la descomposición de tareas de procesamiento complejas en aspectos sencillos cuyos resultados puedan, además, reutilizarse y compartirse en la

composición de otras tareas de procesamiento distintas. En particular, el modelo comparte distintas características con propuestas de modularización de gramáticas de atributos basadas en fragmentos [Kastens & Waite 1992], diferenciándose de aquellas en la posibilidad de utilizar una gramática diferente para cada fragmento. Así mismo, el modelo también tiene características en común con propuestas como la de las gramáticas de atributos fusionables [Farrow et al. 1992], aunque aquéllas están más orientadas a formar especificaciones convencionales gracias a la combinación de subgramáticas componente mediante gramáticas pegamento. Las gramáticas multivista, por el contrario, propugnan una descomposición aspectual del problema, y fomentan el uso de las estructuras sintácticas más adecuadas para cada aspecto.

En este capítulo se han desarrollado, además, métodos específicos para orquestar las dos fases principales involucradas en la ejecución de una gramática de atributos multivista:

- En relación con la fase de análisis, se ha desarrollado MVLR un método de análisis simultáneo con respecto a *n* gramáticas incontextuales conformes entre sí, que se basa en *n* análisis LR sincronizados (la sincronización se realiza durante la reducción a los símbolos de núcleo, y durante los desplazamientos). También se ha estudiado la posibilidad de orquestar el análisis mediante un método LR generalizado (GLR) que opere sobre la gramática global que resulta de unir las *n* gramáticas. Para ello, se ha adaptado el método GLR clásico de Tomita con heurísticas que mejoran el empaquetado de los nodos, a fin de permitir generar los bosques sintácticos normados por las gramáticas multivista. Utilizando una heurística consistente en compartir el mismo nodo de bosque en los arcos creados por la misma acción de reducción, así como otra que lleva a cabo el empaquetado de nodos basado en las profundidades relativas de los mismos, se han conseguido muy buenos resultados, funcionando el método en la práctica totalidad de los casos realistas. No obstante, y debido al carácter heurístico de estas adaptaciones, no ha sido posible garantizar el correcto funcionamiento de las mismas en todos los casos. Sí ha sido posible, no obstante, aprovechar la experiencia ganada para reformular el algoritmo MVLR para trabajar sobre un grafo de pilas (GSS) en lugar de sobre *n* pilas independientes. El método resultante, que se denomina MVGLR, se basa además en el concepto de *multiautómata* LR, la estructura que resulta de representar conjuntamente, mediante un único grafo, todos los autómatas LR de todas las vistas. De esta forma, utilizando dicho multiautómata, es posible generar unas tablas de análisis unificadas que, además de poder servir para orquestar el análisis con respecto a cada gramática individual (de hecho, el método MVLR puede funcionar también guiado por este tipo de tablas), también explicita los estados compartidos, lo que facilita la realización de las bifurcaciones y las reunificaciones de las pilas.

- En relación con la fase de evaluación semántica, se ha adaptado un método de evaluación bajo demanda para que funcione directamente sobre los bosques de análisis sintáctico atribuidos.

Las gramáticas de atributos multivista y los métodos de análisis y evaluación semántica asociados, junto con aspectos de implementación tales como la integración con un marco genérico de procesamiento XML (en concreto, con SAX), así como la configuración de los algoritmos de análisis para que construyan los bosques de análisis atribuidos requeridos por la posterior maquinaria de evaluación semántica, se han plasmado en XLOP3, un entorno para el desarrollo de aplicaciones de procesamiento XML mediante gramáticas de atributos multivista que muestra la viabilidad del enfoque propuesto.

Por último, se ha utilizado XLOP3 en el desarrollo de <eMU>, un sistema para la generación de videojuegos educativos a partir de documentos XML que constituye un caso de estudio realista, de complejidad no trivial, que arroja evidencia positiva respecto a la utilidad en la práctica del enfoque.

Capítulo 6
Conclusiones y Trabajo Futuro

6.1 Introducción

Los capítulos anteriores han descrito el enfoque metalingüístico de desarrollo de aplicaciones de procesamiento XML propuesto en esta tesis. Dicho enfoque propugna el uso de formalismos gramaticales para el desarrollo declarativo de este tipo de aplicaciones, y se encuadra dentro de la línea de investigación en procesamiento dirigido por lenguajes de documentos XML llevada a cabo en el grupo ILSA. En esta tesis se han expuesto de manera unificada los trabajos desarrollados en dicha línea en el seno del grupo, y se han identificado y resuelto dos de sus principales limitaciones: (i) la falta de un criterio para formular gramáticas específicas para el procesamiento conformes con las gramáticas documentales que definen los lenguajes de marcado, y (ii) la falta de un soporte adecuado para la modularización de las especificaciones, que, una vez descompuesta una tarea de procesamiento en subtareas, permita utilizar sintaxis específicas apropiadas para cada subtarea, posiblemente distintas entre sí.

De esta forma, el presente capítulo concluye esta memoria de tesis, resumiendo las principales aportaciones (sección 6.2) y, describiendo algunas líneas de investigación futura (sección 6.3).

6.2 Principales Aportaciones

Esta sección se resume las principales aportaciones realizadas en esta tesis. Dichas aportaciones pueden, a su vez, clasificarse en términos de los objetivos inicialmente planteados.

6.2.1 Aportaciones relativas a la caracterización del desarrollo de aplicaciones de procesamiento XML como una actividad metalingüística

En relación con el primer objetivo planteado, la caracterización del desarrollo de aplicaciones de procesamiento XML como actividad metalingüística, cabe destacar las siguientes aportaciones:

- Análisis unificado de los enfoques dirigidos por lenguajes al procesamiento de documentos XML llevados a cabo en el grupo de investigación ILSA de la UCM, en términos de modelos de proceso y formalismos declarativos de especificación.

- Identificación de la técnica basada en cambios de estado como aspecto clave para la aplicación práctica de las gramáticas de atributos a la especificación de tareas de procesamiento de documentos XML.

- Identificación de las dos principales limitaciones de los enfoques al desarrollo dirigido por lenguajes propuestos en el grupo ILSA.

A continuación, se describen con más detalle cada una de estas aportaciones.

6.2.1.1 Análisis unificado de los enfoques dirigidos por lenguajes al desarrollo de aplicaciones XML

En esta tesis se ha realizado un análisis unificado de los distintos enfoques al desarrollo dirigido por lenguajes de aplicaciones de procesamiento XML que se han formulado en el grupo de investigación ILSA, enfoques en cuya formulación y maduración, el autor de esta tesis también ha participado activamente. El análisis realizado pone de manifiesto que:

- Dichos enfoques adaptan modelos de procesos clásicos seguidos en el diseño y la implementación de lenguajes informáticos al dominio del procesamiento de la documentación XML. Este hecho es cierto tanto en lo que se refiere al desarrollo basado en herramientas convencionales de construcción de procesadores de lenguaje (herramientas que, como JavaCC o CUP, soportan esquemas de traducción), como al desarrollo basado en formalismos declarativos de más alto nivel (como es el caso del uso de gramáticas de atributos y del sistema XLOP).

- En todos ellos juega un papel central la formulación de una gramática incontextual que caracteriza estructuralmente el lenguaje de marcado de cara al procesamiento y que, por tanto, es diferente a la gramática documental utilizada en la definición del mismo.

- Así mismo, todos ellos propugnan el posterior añadido de semántica a dicha gramática específica para el procesamiento (acciones semánticas en el caso de esquemas de traducción, ecuaciones semánticas en el caso de las gramáticas de atributos).

- De la misma forma, todos estos enfoques propugnan una separación explícita entre la lógica específica de la aplicación (normalmente proporcionada mediante un marco de aplicación específico), y la capa lingüística encargada de llevar a cabo el procesamiento de los documentos. La mediación entre ambos niveles se lleva a cabo a través de los servicios de una *clase semántica* (que implementa las operaciones básicas a utilizar desde los esquemas de traducción, o las funciones semánticas a integrar en las gramáticas de atributos).

- Todos estos enfoques adoptan una estrategia generativa para el desarrollo, mantenimiento y evolución de la capa lingüística. Esta estrategia propugna la descripción a alto nivel de dicha capa, utilizando un formalismo específico orientado a la caracterización de tareas de procesamiento dirigido por la sintaxis, así como una

meta-herramienta que soporte dicho formalismo (p.e., herramientas convencionales de generación de traductores, o meta-meta herramientas como XLOP).

- Por último, todos estos enfoques propugnan la integración de los procesadores generados con un marco de procesamiento XML de propósito general convencional, que, en este contexto, juega el mismo papel que el de los analizadores léxicos en un procesador de lenguaje prototípico.

Estos hechos se recogen en trabajos como [Sarasa et al. 2009-a], en el que se ejemplifica cómo amalgamar un marco genérico para el procesamiento de documentos XML (StAX) con una herramienta de generación de traductores (CUP), o [Sarasa et al. 2012], en el que se abstrae el proceso genérico subyacente a este tipo de combinación. Los hechos se recogen también en [Sarasa et al. 2009-b, Sarasa et al. 2011], donde se describe la meta-meta herramienta basada en gramáticas de atributos XLOP.

6.2.1.2 La técnica de cambios de estado para las especificaciones basadas en gramáticas de atributos de tareas de procesamiento de documentos XML

En relación con la especificación práctica de tareas de procesamiento basadas en gramáticas de atributos, el autor ha constatado que, en la práctica, es posible concebir la clase semántica que proporciona las funciones semánticas, no únicamente como una colección de funciones, sino también como una representación explícita del estado del procesamiento. De esta forma, mediante el almacenamiento explícito de estado en la instancia de la clase semántica, es posible simplificar substancialmente las especificaciones. Bajo este supuesto, existirán atributos que no representarán directamente valores computados, sino cambios de estado. Esta técnica es análoga a la sugerida, por ejemplo, por [Kastens et al. 1994] en relación con la combinación de gramáticas de atributos y tipos abstractos de datos.

El autor ha aplicado con éxito esta técnica en el desarrollo con XLOP del generador para <e-Tutor>, tal y como se describe en [Temprado et al. 2010-b].

6.2.1.3 Identificación de las principales limitaciones de los enfoques dirigidos por lenguajes del grupo ILSA

El análisis realizado sobre los enfoques dirigidos por lenguajes desarrollados en el grupo ILSA ha puesto también de manifiesto dos de las principales limitaciones anticipadas para este tipo de enfoques:

- Por una parte, los trabajos realizados han ignorado la relación existente entre gramáticas específicas para el procesamiento y gramáticas documentales. Efectivamente, ambas descripciones deben cumplir algún criterio de equivalencia. No obstante, la formulación y comprobación de dicho criterio han sido ignorados

sistemáticamente en los trabajos previos realizados, dejando la satisfacción de los mismos a manos del desarrollador, con los problemas y errores que esta práctica puede acarrear.

- Por otra parte, los trabajos realizados han ignorado, en su mayor parte, las cuestiones relativas a la modularidad de las especificaciones de la capa lingüística. Dicha modularidad es necesaria para casos de complejidad no trivial, que involucren tareas de procesamiento complejas que pueden, a su vez, descomponerse en tareas de procesamiento más simples. Una excepción a este respecto son los trabajos descritos en [Sarasa et al. 2008, Sarasa & Sierra 2013-b]. No obstante, dichos trabajos adolecen de un problema común: las distintas tareas de procesamiento deben formularse sobre una gramática común, independientemente de donde las estructuras modelizadas por dicha gramática sean convenientes o no para cada subtarea.

Estas limitaciones están presentes tanto en los enfoques basados en esquemas de traducción [Sarasa et al. 2009-a, Sarasa et al. 2012], como en los enfoques basados en gramáticas de atributos [Sarasa et al. 2009-b, Temprado et al. 2010-b, Sarasa et al. 2011].

6.2.2 Aportaciones relativas a la caracterización de la conformidad entre gramáticas y a su comprobación automática

En relación con el segundo objetivo planteado, la caracterización de la conformidad entre gramáticas documentales y gramáticas específicas del procesamiento y formulación de un método efectivo para comprobar dicha conformidad, se destacan las siguientes aportaciones:

- Propuesta de un método sistemático para la formulación de gramáticas incontextuales específicas para el procesamiento dirigido por lenguajes de documentos XML.
- Propuesta de un método efectivo y automatizable para la comprobación de la conformidad de dichas gramáticas con respecto a gramáticas EBNF de base.
- Generalización del método de la comprobación de la conformidad para abordar la conformidad entre dos gramáticas BNF arbitrarias.

6.2.2.1 Método sistemático para la formulación de gramáticas incontextuales específicas para el procesamiento

En esta tesis se ha propuesto un método sistemático para la formulación de las gramáticas específicas para el procesamiento, que parte de la abstracción de las características estructurales básicas de la gramática documental mediante una gramática EBNF. Entonces propugna la *realización* de dicha gramática proporcionando para cada no terminal de la misma una subgramática no autoembebible que *implementa* la expresión regular asociada [Temprado

et al. 2010-a]. La elección de la clase de gramáticas no autoembebibles para realizar las expresiones regulares EBNF obedece a las siguientes consideraciones:

- Por una parte, la clase de gramáticas no autoembebibles no constriñe la forma de las gramáticas, más allá de preservar la condición de no autoembebilidad (es decir, asegurar que las recursiones siempre ocurren en los extremos, pero nunca en el interior de las formas sentenciales generadas). De esta forma, se dispone de flexibilidad suficiente como para proporcionar caracterizaciones BNF naturales para las expresiones regulares presentes en la gramática EBNF de base.

- Por otra parte, las gramáticas de dicho tipo siempre generan lenguajes regulares. De hecho, existen métodos que permiten su transformación inmediata a descriptores convencionales para dichos lenguajes (p.e., AFNDs). Esto facilitará la posterior formulación del criterio de conformidad, así como la automatización de su comprobación.

De esta forma, la gramática específica para el procesamiento será una realización de la gramática EBNF de base que realiza gramaticalmente (mediante gramáticas no autoembebibles) cada una de las definiciones en dicha gramática. El proceso de formulación en sí es de carácter iterativo e incremental, ya que tanto la gramática EBNF de base como la gramática BNF específica para el procesamiento pueden refinarse conforme se descubren violaciones en la conformidad.

6.2.2.2 Método automático para la comprobación de la conformidad

Las restricciones impuestas en la formulación de las gramáticas específicas para el procesamiento permiten reducir el problema de la comprobación de la conformidad a *n* problemas de comprobación de la equivalencia de una gramática no autoembebible con respecto a una expresión regular. Dichos problemas son automatizables utilizando los distintos elementos de la teoría de lenguajes formales analizados en la memoria.

En particular, en esta tesis hemos prestado especial atención al factor humano en el proceso de comprobación de la conformidad. Para ello, hemos elegido un método para la comprobación de la equivalencia entre gramáticas no autoembebibles y expresiones regulares que permita proporcionar información lo más precisa posible en caso de detección de no equivalencia. Para tal fin:

- Hemos adoptado un método basado en *derivadas parciales* de expresiones regulares [Antimirov 1996] para transformar expresiones regulares en AFNDs equivalentes. Dicho método caracteriza explícitamente el lenguaje aceptado desde cada estado del AFND mediante una expresión regular (la correspondiente derivada parcial) asociada al mismo. Por consiguiente, la determinización de este autómata genera estados a su vez etiquetados con expresiones regulares, y, por tanto, susceptibles de proporcionar información concisa sobre el lenguaje que queda por reconocer, en lo que respecta a la expresión regular, en caso de detectar una no equivalencia.

- Para poder aprovechar esta característica también en el caso de las gramáticas no autoembebibles, hemos desarrollado un algoritmo novedoso para transformar dichas gramáticas en expresiones regulares equivalentes, refinando la propuesta descrita en [Temprado et al. 2011]. Al contrario que propuestas como [Nederhof 2000], dicho algoritmo, combinado con el método de las derivadas parciales y la posterior determinización del AFND resultante, permitirá ofrecer también información de diagnóstico más precisa en lo que se refiere a la parte de la gramática no autoembebible.

El método resultante es susceptible, así mismo, de ser implementado perezosamente, tanto en lo que se refiere a la comprobación de la equivalencia (que, finalmente, radica en comprobar que los AFDs resultantes de las determinizaciones son equivalentes entre sí), como en lo que se refiere a la traducción de expresiones regulares en AFNDs equivalentes.

6.2.2.3 Generalización del método para la comprobación de la conformidad

En esta tesis hemos explorado, además, la generalización del método para la comprobación de la conformidad a gramáticas BNF arbitrarias. Para tal fin, nos hemos percatado de que el método de traducción de gramáticas no autoembebibles a expresiones regulares puede adaptarse con unas modificaciones mínimas para que opere sobre gramáticas arbitrarias. En este caso, el resultado no es ya una expresión regular, sino una gramática EBNF [Temprado et al. 2011]. Esto permite comprobar la conformidad entre gramáticas BNF arbitrarias mediante:

- Su transformación a gramáticas EBNF equivalentes, aplicando la generalización del método de transformación de gramáticas no autoembebibles en expresiones regulares.

- La comprobación de que dichas gramáticas involucran los mismos no terminales, y que las correspondientes definiciones son equivalentes una a una. Dicha comprobación puede realizarse, por ejemplo, utilizando el algoritmo de las derivadas parciales, más la construcción por subconjuntos [Aho et al. 2006], más un método de comprobación de la equivalencia entre AFDs (p.e., [Hopcroft & Karp 1971]).

Esta generalización ha dado lugar a la construcción de una herramienta denominada *Grammar Equivalence Checker*. Dicha herramienta ha sido, además, evaluada con éxito en el contexto de uso de un curso de Procesadores de Lenguaje en la Facultad de Informática de la UCM, donde la herramienta ha resultado útil durante la transformación sistemática de gramáticas, así como ha sido bien aceptada por parte de los alumnos (alumnos de último curso, y, por tanto, desarrolladores en el futuro más inmediato).

6.2.3 Aportaciones relativas a la mejora de los mecanismos de la modularidad de las especificaciones

En relación con el tercero de los objetivos planteados, la mejora de los mecanismos de modularidad en la descripción metalingüística de aplicaciones de procesamiento de documentos XML, se destacan las siguientes aportaciones:

- Propuesta de un nuevo método de modularización de gramáticas de atributos, las gramáticas de atributos multivista, apropiado para abordar la especificación modular de tareas de procesamiento XML.

- Propuesta de nuevos métodos de análisis sintáctico, adecuados para abordar el análisis simultáneo de un documento con respecto a *n* gramáticas conformes entre sí.

- Propuesta de un método de evaluación semántica que opera sobre bosques de análisis sintáctico atribuidos.

Los siguientes puntos presentan con más detalle cada una de estas aportaciones.

6.2.3.1 Propuesta del modelo de las gramáticas de atributos multivista

En esta tesis se ha propuesto un nuevo modelo de gramáticas de atributos, las *gramáticas de atributos multivista* [Temprado et al. 2010-a], que proporciona un soporte más adecuado a la descomposición jerárquica de tareas de procesamiento XML en subtareas. Las principales características del modelo son:

- Las especificaciones se organizan en términos de *vistas gramaticales*. Cada una de estas vistas puede utilizar una sintaxis específica para la vista. Desde un punto de vista semántico, se conciben como fragmentos de gramática de atributos.

- Las gramáticas incontextuales subyacentes a las vistas deben ser conformes con una gramática EBNF de base. De esta forma, todas estas gramáticas comparten un conjunto de símbolos no terminales de núcleo.

- Desde un punto de vista sintáctico, las diferentes gramáticas incontextuales subyacentes a las vistas (que deben de ser no ambiguas) inducen sobre cada documento un árbol sintáctico diferente. Todos estos árboles se pueden unir a través de los símbolos de núcleo para dar lugar a un bosque de análisis sintáctico.

- Desde un punto de vista semántico, el proceso de evaluación semántica se lleva a cabo sobre el bosque de análisis sintáctico. En cada vista es posible consultar los atributos de los símbolos de núcleo definidos en otras vistas (heredados, en caso de que el símbolo de núcleo ocurra en posiciones de cabeza de producción, sintetizados, en caso de que ocurra en posiciones de cuerpo). Esta característica permite interrelacionar las vistas entre sí.

El modelo, por tanto, comparte características con las propuestas a la modularización de gramáticas de atributos basadas en fragmentos, cada uno de los cuáles resuelve un determinado aspecto semántico [Kastens & Waite 1992]. No obstante, al contrario que dichas propuestas, las gramáticas de atributos permiten utilizar sintaxis diferentes en cada fragmento, a fin de adecuar dichas sintaxis a las necesidades de procesamiento. El modelo también es similar al de las gramáticas de atributos fusionables [Farrow et al. 1992]. No obstante, mientras que aquellas están orientadas a crear especificaciones basadas en gramáticas de atributos convencionales mediante la unión de gramáticas componente con gramáticas pegamento, el modelo de las gramáticas multivista está más orientado a establecer diferentes perspectivas de una misma especificación, perspectivas que, posteriormente, se realizan mediante una sintaxis y una semántica específica. Este enfoque permite, por tanto, utilizar gramáticas de procesamiento específicas para cada una de las perspectivas, eliminando, de esta forma, la restricción de tener que adoptar una gramática incontextual común para todos los aspectos del procesamiento.

En esta tesis se ha demostrado, además, la viabilidad del enfoque mediante la construcción del entorno XLOP3, una evolución del entorno XLOP que soporta el modelo de las gramáticas de atributos multivista. Así mismo, se ha demostrado también la potencia del mismo mediante el desarrollo con XLOP3 de <eMU>, un sistema de complejidad no trivial que integra un lenguaje específico de dominio basado en XML orientado a la descripción documental de juegos educativos, así como un generador de videojuegos dirigido por dicho lenguaje (componente que ha sido el desarrollado con XLOP3).

6.2.3.2 Propuesta de métodos de análisis multivista

La implementación eficiente del modelo de las gramáticas de atributos multivista necesita resolver el problema del análisis simultáneo de un documento respecto a *n* gramáticas incontextuales conformes entre sí [Temprado et al. 2010-a]. Para ello, en esta tesis:

- Se ha estudiado, por una parte, la viabilidad de adaptar métodos de análisis sintáctico generalizados (en particular, métodos GLR) para dicho fin. Para ello se ha partido de los métodos básicos propuestos por Tomita [Tomita 1985, Tomita 1986], y se han ensayado distintas heurísticas (compartición del mismo nodo de bosque en los arcos creados por la misma acción de reducción, empaquetado de nodos basado en las profundidades relativas de los mismos) que han arrojado resultados muy satisfactorios. No obstante, y debido al carácter heurístico de estas adaptaciones, no ha sido posible garantizar el correcto funcionamiento de las mismas en todos los casos.

- Se ha diseñado un nuevo método, MVLR, basado en el desarrollo simultáneo de *n* procesos de análisis LR que se sincronizan a través de las reducciones por símbolos de núcleo. Al contrario que la anterior opción, MVLR sí funciona correctamente para todos los casos.

- Tomando como punto de partida la idea básica de los algoritmos GLR, se ha rediseñado MVLR para que opere sobre un grafo de pilas (una GSS), en lugar de sobre *n* pilas independientes entre sí. El método resultante, que se ha denominado MVGLR, depende del concepto de *multiautómata* LR: representación sobre una misma estructura de los *n* autómatas LR asociados con las *n* vistas. De esta forma, el multiautómata LR explicita las partes en común que tienen los distintos autómatas asociados con las distintas vistas, y permite, así mismo, generar unas tablas de análisis LR unificadas, que son la base para gestionar adecuadamente el grafo de pilas. Para ello, inicialmente se parte de los estados iniciales asociados con cada vista. Las pilas, entonces, pueden reunificarse en los desplazamientos y en las reducciones por símbolos de núcleo, y pueden bifurcarse en las reducciones en general. Por lo demás, el algoritmo MVGLR presenta una estructura y comportamiento equivalente al MVLR (y, de hecho, el MVLR puede operar con las tablas unificadas generadas a partir del multiautómata, en lugar de con las *n* tablas asociadas con las *n* vistas).

En esta tesis se ha mostrado, además, cómo cualquiera de los métodos anteriores (en particular, el MVLR y el MVGLR) pueden integrarse con marcos de procesamiento XML genéricos (en concreto, se ha estudiado la integración con SAX), satisfaciendo, de esta forma, uno de los invariantes detectados en el análisis de los enfoques dirigidos por lenguajes previos desarrollados en el grupo ILSA. También se ha mostrado cómo pueden instrumentarse para que, en lugar de producir bosques de análisis sintáctico, realicen también la atribución de dichos bosques a fin de posibilitar la posterior fase de evaluación semántica. Por último, y a través de la integración de estos métodos en el núcleo de análisis de XLOP3, se ha mostrado la viabilidad de su aplicación práctica.

6.2.3.3 Propuesta de métodos de evaluación semántica multivista

En esta tesis se ha desarrollado, por último, un modelo de evaluación semántica que, en lugar de operar sobre los árboles de análisis sintáctico, opera sobre los bosques de análisis sintáctico generados por el modelo de las gramáticas multivista durante el procesamiento de un documento. Dicho modelo adapta un método de evaluación semántica dinámica bajo demanda convencional sobre árboles [Magnusson & Hedin 2007]. El método se ha implementado satisfactoriamente, así mismo, en el sistema XLOP3, lo que muestra de nuevo la viabilidad de la propuesta.

6.3 Líneas futuras de investigación

En esta sección formulamos, por último, algunas líneas futuras de investigación que pueden desprenderse de esta tesis. Más concretamente, se considera interesante continuar investigando en las tres líneas siguientes:

- *Refinamiento del enfoque*. Esta línea desarrollará diferentes aspectos del enfoque que no se han tratado en esta tesis.

- *Evaluación empírica del enfoque*. Esta línea estará orientada a evaluar empíricamente el enfoque, mediante la realización de diferentes experimentos.

- *Aplicación del enfoque a otros dominios de gestión de la información*. Esta línea estará orientada a extrapolar el enfoque a otros dominios diferentes al del procesamiento de documentos XML.

A continuación, se desarrollan con más detalle cada una de estas líneas.

6.3.1 Refinamiento del enfoque

En esta tesis se han abordado tres aspectos claves del enfoque metalingüístico al procesamiento de documentos XML: (i) análisis unificador de los trabajos previos realizados, (ii) conformidad de gramáticas específicas para el procesamiento con respecto a gramáticas documentales, y (iii) mejora de los aspectos de modularización de las especificaciones. Para ello se han abordado las cuestiones que se han considerado más esenciales, y se han relegado otras que se han considerado menos críticas, pero que, sin embargo, merecen también ser tratadas con la debida profundidad. En particular, pueden resaltarse los siguientes como aspectos que merecen un esfuerzo adicional de investigación en el futuro:

- Mejor integración con formalismos de gramáticas documentales. En esta tesis, con el fin de abstraer el problema de la conformidad de formalismos de gramáticas documentales concretos, se ha propuesto utilizar gramáticas EBNF como representación intermedia capaz de abstraer las características estructurales esenciales de los lenguajes de marcado objeto del procesamiento. No obstante, no se ha profundizado en la forma de llevar a cabo la formulación sistemática de dichas gramáticas, sino que se ha relegado su formulación al criterio y a la experiencia del desarrollador. Desde este punto de vista, se considera interesante profundizar, por una parte, en el problema de abstracción EBNF de gramáticas documentales. Dicho problema será específico de cada formalismo gramatical concreto, aunque, probablemente, puedan identificarse patrones generalizables a diferentes casos. Idealmente, el proceso de obtención de la gramática debería poder ser describible en términos de pasos de transformación correctos, trazables y automatizables. Así mismo, dicho proceso de integración debería tener en cuenta otros aspectos que no se han tratado en esta tesis (atributos de los elementos, espacios de nombres, tipado de contenidos y valores de atributos, etc.).

- Estudio de la conjetura para los autómatas LALR(1) de las distintas vistas. Durante el desarrollo de las propuestas de análisis ha surgido, como propuesta central, la de *multiautómata* LR. Mientras que en caso de autómatas LR(0) o LR(1) los estados compartidos en el multiautómata son idénticos en las vistas implicadas, en el caso de autómatas LALR(1), los estados compartidos *fusionan* los estados de los autómatas individuales (entendida la fusión como la unificación en un único estado de la parte LR(0) común, uniendo los símbolos de preanálisis para cada uno de los elementos). En

esta tesis hemos conjeturado, así mismo, que, también en este caso, los estados compartidos son idénticos en todas las vistas (es decir, coinciden tanto en el corazón como en la correspondiente información de preanálisis), por lo que en el multiautómata no se generarán, tampoco en este caso, nuevos conflictos. No obstante, no hemos proporcionado una prueba formal de dicha conjetura. Es por tanto que proponemos llevar a cabo un estudio más profundo de la misma, construyendo la citada prueba, o bien construyendo un contraejemplo.

- Optimización del método MVGLR. En la formulación del método MVGLR no hemos hecho especial énfasis en una gestión eficiente de la GSS subyacente. En particular, con el fin de permitir escalar el enfoque al proceso de documentos grandes, sería interesante estudiar mecanismos de liberación de basura, que permitieran liberar las partes de la GSS que ya no se necesitan, con el fin de mantener limitado el tamaño de la misma. También sería interesante idear mecanismos que evitaran el uso de la memoria dinámica para representar la GSS, manteniendo, por ejemplo, un número fijo de pilas (dado que, a lo sumo, existirá un máximo de *n* pilas abiertas en cada momento) y un protocolo que permita mapear la GSS sobre dichas pilas.

- Optimización de la fase de evaluación semántica. El método de evaluación semántica bajo demanda propuesto implica la construcción explícita del SPPF. A este respecto, y para mejorar la escalabilidad de la propuesta, así como para permitir su aplicación a escenarios en los que los documentos se procesan conforme estos se generan (p.e., canales de comunicación estructurados mediante XML), se considera interesante estudiar mecanismos alternativos que permitan optimizar dicha fase de evaluación. A este respecto, se considera interesante adoptar una estrategia dirigida por los datos, como la utilizada en XLOP, que, al mismo tiempo, evite la construcción explícita del SPPF, propugnando, en su lugar, la construcción incremental del grafo de dependencias entre atributos (así como mecanismos eficientes para llevar a cabo la liberación de la basura generada en el proceso).

6.3.2 Evaluación empírica del enfoque

En esta tesis se ha mostrado la viabilidad del enfoque basado en las gramáticas de atributos multivista mediante la construcción de XLOP3 y mediante su aplicación al desarrollo de <eMU>, un caso de estudio de complejidad no trivial. Se ha realizado, así mismo, un experimento inicial de la eficacia y usabilidad del método de comprobación de la conformidad entre gramáticas. No obstante, se considera interesante llevar a cabo también esfuerzos sistemáticos para una evaluación empírica más profunda del enfoque. Dichos esfuerzos se centrarán en comparar sistemáticamente el enfoque metalingüístico propuesto en esta tesis con otros enfoques de desarrollo más convencionales. En particular, dichas comparaciones estarán orientadas a la:

- Comparación sistemática de las soluciones. El objetivo de este esfuerzo es comparar el enfoque metalingüístico propuesto en esta tesis con enfoques más convencionales al

desarrollo de aplicaciones de procesamiento XML. Para ello se considera interesante diseñar una batería significativa de tareas de procesamiento de documentos XML y, seguidamente, proporcionar soluciones basadas en enfoques convencionales (p.e., uso de marcos genéricos como DOM, SAX, o StAX, o enfoques específicos como XSLT), así como basadas en XLOP3. Esto permitirá, entonces, comparar las soluciones en base a criterios estáticos tanto objetivos (p.e., líneas de código), como subjetivos (p.e., claridad y legibilidad de la solución), como dinámicos (p.e., tiempo de proceso sobre diferentes documentos, memoria consumida).

- Comparación sistemática de los procesos de desarrollo. A este respecto pueden plantearse estudios experimentales basados en grupos experimentales (que utilicen XLOP3) y grupos de control (que utilicen otras técnicas) a los que se les propone la solución de las distintas tareas de procesamiento diseñadas. Durante el desarrollo de las distintas tareas pueden medirse distintos parámetros (tiempo invertido, errores cometidos, etc.) que sirvan para comparar el enfoque propuesto con los más convencionales desde el punto de vista de su eficacia para el desarrollo. Estos experimentos pueden tener en cuenta también factores relativos al mantenimiento y a la evolución de las aplicaciones de procesamiento, a fin de contrastar las posibles ventajas de la propuesta planteada en lo que se refiere a estos aspectos críticos del ciclo de vida.

- Evaluación de la satisfacción por parte de los desarrolladores. Este esfuerzo debe enfocarse a medir, a través de diferentes dimensiones, en qué grado la propuesta puede ser bien acogida por la comunidad de desarrolladores, así como a evidenciar también posibles carencias del enfoque y posibles formas de abordar las mismas.

6.3.1 Aplicación a otros dominios

Como tercera línea de investigación se plantea, por último, extrapolar el enfoque metalingüístico presentado en esta tesis a otros dominios de gestión de información. A este respecto, se considera interesante explorar el uso de gramáticas de atributos multivista en campos a los que ya se ha extrapolado el uso de gramáticas de atributos en el grupo ILSA. En concreto:

- Procesamiento de datos JSON. Efectivamente, en [Sarasa & Sierra 2013-a] se describe el uso de gramáticas de atributos para especificar declarativamente dichas tareas de procesamiento.

- Procesamiento de redes de objetos Java. En [Sarasa & Sierra 2015] se describe una propuesta de uso de gramáticas de atributos para especificar declarativamente el procesamiento de grafos de objetos Java arbitrarios.

Así mismo, se considera interesante explorar la aplicación del enfoque a otro tipo de problemas de procesamiento de la información (procesamiento de grafos RDF, procesamiento de secuencias de eventos en aplicaciones interactivas, etc.).

Referencias

[Abiteboul et al. 2000] Abiteboul, S., Buneman, P., Suciu, D. "Data on the Web: From Relations to Semistructured Data and XML". Morgan Kaufmann. 2000.

[Adams 1991] Adams, S. "Modular Attribute Grammars for Programming Language Prototyping". PhD. Thesis. University of Southampton. 1991.

[Aho et al. 2006] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D. "Compilers: Principles, Techniques and Tools". 2nd Edition. Addison-Wesley. 2006.

[Akker et al. 1991] Akker, R., Melichar, B., Tarhio, J. “Attribute Evaluation and Parsing”. Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science, Springer, 545: 187-214. 1991.

[Alblas 1991] Ablas, H. “Attribute Evaluation Methods”. Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science, 545: 48-113. 1991.

[Almeida et al. 2010] Almeida M., Moreira N., Reis R. "Testing the Equivalence of Regular Languages". Journal of Automata, Languages and Combinatorics, 15(1/2): 7-25. 2010.

[Andrei et al. 2003] Andrei, S., Cavadini, S.V., Chind, W.N. "A New Algorithm for Regularizing One-letter Context-free Grammars". Theoretical Computer Science 306(1-3):113–122. 2003.

[Andrei et al. 2004] Andrei, S., Chind, W-N., Cavadini, S. “Self-embedded Context-free Grammars with Regular Counterparts”. Acta Informatica 40(5): 349–365. 2004.

[Antimirov 1996] Antimirov, V. “Partial Derivatives of Regular Expressions and Finite Automaton Constructions”. Theoretical Computer Science, 155(2): 291-319. 1996.

[Appel 1997] Appel, A.W. “Modern Compiler Implementation in Java”. Cambridge University Press. 1997.

[Baeten et al. 1993] Baeten, J., Bergstra, J., Willem, J. "Decidability of Bisimulation Equivalence for Processes Generating Context-free Languages". Journal de the ACM 40(3): 653-682. 1993.

[Bar-Hillel et al. 1961] Bar-Hillel, Y., Perles, M., Shamir, E. “On Formal Properties of Simple Phrase-structure Grammars”. Z. Phonetik, Sprachwiss, Kommunikationsforsch, 14: 143-172. 1961.

[Berry & Sethi 1986] Berry, G., Sethi, R. "From Regular Expressions to Deterministic Automata". Theoretical Computer Science 48(3), 117-126. 1986.

[Birceck et al. 2001] Birbeck, M., Duckett, J., Gudmundsson, O.G., Kobak, P., Lenz, E., Livingstone, S., Marcus, D., Mohr, S., Pinnock, J., Visco, K., Watt, A., Willians, K., Zaev, Z., Ozu, N. "Professional XML 2nd Edition". Wrox Press. 2001.

[Biron et al. 2001] Biron, P.V., Malhotra, A. "XML Schema Part 2: Datatypes". W3C Recommendation. 2001.

[Bradley 2001] Bradley, N. “The XML Companion (3rd Edition)”. Addison-Wesley. 2001.

[Bray et al. 2008] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F. “Extensible Markup Language (XML) 1.0 (Fifth Edition)”. W3C Recommendation. 2008.

[Brownell 2002] Brownell, D. “SAX2 - Procesing XML Efficiently with Java”. O’Relley. 2002.

[Brüggemann-Klein 1993] Brüggemann-Klein, A. "Regular Expressions into Finite Automata". Theoretical Computer Science. Volume 120, Issue 2, 22. pp:197–213. 1993.

[Brüggemann-Klein et al. 1998] Brüggemann-Klein, A., Wood, D. "One-unambiguous Regular Languages". Information and Computation, 142(2), pp. 182-206. 1998.

[Brzozowski 1964] Brzozowski, J. “Derivatives of Regular Expressions”. Journal of the ACM, 11(4): 481-494. 1964.

[Champarnaud & Ziadi 2002] Champarnaud, J.M., Ziadi, D. "Canonical Derivatives, Partial Derivatives and Finite Automaton Constructions". Theoretical Computer Science. Volume 289, Issue 1, 23 October, pp:137–163. 2002.

[Chomsky 1956] Chomsky, N. "Three Models for the Description of Language". IRE Transactions on Information Theory, 2(3):113-124. 1956.

[Chomsky 1957] Chomsky, N. "Syntactic Structures". 's-Gravenhage: Mouton & Co., N.V.:116. 1957.

[Clark 1999] Clark, J. "XSL Transformations (XSLT) Version 1.0". W3C Recommendation. 1999.

[Clark 2001-a] Clark, J. "TREX – Tree Regular Expressions for XML Language Specification". Thai Open Source Software Center. 2001.

[Clark 2001-b] Clark, J. "TREX – Tree Regular Expressions for XML Tutorial". Thai Open Source Software Center. 2001.

[Clark et al. 1999] Clark, J., DeRose, S. "XML Path Language (XPath) Version 1.0". W3C Recommendation. 1999.

[Clark et al. 2001-a] Clark, J., Makoto, M. "Relax NG Specification". Oasis Committee Specification. 2001.

[Clark et al. 2001-b] Clark, J., Makoto, M. "Relax NG Tutorial". Oasis Committee Specification. 2001.

[Cleaveland 2001] Cleaveland, J. “Program Generators with XML and Java”. Prentice Hall. 2001.

[Cohen & Harry 1979] Cohen, R., Harry, E. “Automatic Generation of Near-optimal Linear-time Translators for Non-circular Arttribute Grammars”. 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages: 121-134. 1979.

[Comon et al. 1997] Comon, H., Dauchet, M., Gilleron, R., Jacquermard, F., Lugiez, D., Tison, S., Tommasi, M. "Tree Automata Techniques and Applications". Technical Report. Groupe de Recherche sur l’Apprentissage Automatique. Univ. Lille. 1997.

[Coombs et al. 1987] Coombs, J. H., Renear, A. H., DeRose, S. J. “Markup Systems and the Future of Scholarly Text Processing”. Communications of the ACM, 30(11): 933-947. 1987.

[Czarnecki & Eisenecker 2000] Czarnecki, K., Eisenecker, U. "Generative Programming: Methods, Tools, and Applications". Addison Wesley, 2000.

[Dabek et al. 2002] Dabek, F., Zeldovich, N., Kaashoek, M. F., Mazières, D., Morris, R. "Event-driven Programming for Robust Software". 10th ACM SIGOPS European Workshop. ACM, pp. 22-25. 2002.

[DeRemer 1969] DeRemer, F. “Practical Translators for LR(k) Languages”. Ph.D. Thesis, MIT, Cambridge, MA, 1969.

[Dilkes & Visnevski 2004] Dilkes, F. A., Visnevski, N. "Non-self-embedding Context-free Grammars for Electronic Warfare". Defence R&D Canada - Ottawa TECHNICAL MEMORANDUM DRDC Ottawa TM 2004-157. October. 2004.

[DOM 2009] “DOM. Document Object Model Technical Reports”. W3C Recommendations. www.w3.org/DOM/DOMTR. 2009.

[Draper et al. 2003] Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M., Siméon, J., Wadler, P. "XQuery 1.0 and XPath 2.0 Formal Semantics". W3C Working Draft. 2003.

[Dueck & Cormack 1990] Dueck, G., Cormack, G. "Modular Attribute Grammars". Computing Journal 33,164-172. 1990.

[Earley 1968] Earley, J. “An Efficient Context Free Parsing Algorithm”. PhD thesis, Computer Science Department, Carnegie-Mellon University. 1968.

[Fallside 2001] Fallside, D.C. "XML Schema Part 0: Primer". W3C Recommendation. 2001.

[Farrow et al. 1992] Farrow, R., Marlowe T.J., Yellin, D.M., “Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation”. 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1992.

[Fischer et al. 2010] Fischer, C.N., Cytron, R.K, LeBlanc, R.J. "Crafting A Compiler: Global Edition". Pearson Higher Education. 2010.

[Fowler 2010] Fowler, M. “Domain Specific Languages”. Addison-Wesley. 2010.

[Gançarski et al. 2002] Gançarski, A. L., Doucet, A., Henriques, P. R. "Information Retrieval from Structured Documents Represented by Attribute Grammars". International Conference on Information Systems Modelling. 2002.

[Gançarski et al. 2006] Gançarski, A. L., Doucet, A., Henriques, P. R. "Attribute Grammar-based Interactive System to Retrieve Information from XML Documents". IEEE Proceedings-Software, 153(2), pp. 51-60. 2006.

[Ganzinger & Giegerich 1984] Ganzinger, H., Giegerich, R. “Attribute Coupled Grammars”. SIGPLAN symposium on Compiler construction, ACM SIGPLAN Notices, 19(6): 157-170. 1984.

[García et al. 2011] García, P. Lópeza, D., Ruiza, J., Álvarez, G.I. "From Regular Expressions to Smaller NFAs". Theoretical Computer Science. Volume 412, Issue 41, 23. pp:5802–5807. 2011.

[Goldfarb 1981] Goldfarb, C. “A Generalized Approach to Document Markup”. ACM SIGPLAN Notices, 16(6): 68-73. 1981.

[Goldfarb 1991] Goldfarb, C. “The SGML Handbook”. Oxford University Press. 1991.

[Grune & Jacobs 2008] Grune, D., Jacobs, C.J.H. "Parsing Tecniques - A Practical Guide". 2nd Edition, Monographs in Computer Science, Springer. 2008.

[Havasi 2002] Havasi, F. “XML Semantics Extension”. Acta Cybernetica, 15(2): 509-528. 2002.

[Hedin 1989] Hedin, G. "An Object-oriented Notation for Attribute Grammars". Proceedings of the European Conference on Object-Oriented Programming ECOOP’89. 1989.

[Hedin 1999] Hedin, G. "Reference Attribute Grammars". Second Workshop of Attribute Grammars and their Applications, WAGA’99. Amsterdam, The Netherlands. 1999.

[Hopcroft & Karp 1971] Hopcroft, J., Karp, R. “A Linear Algorithm for Testing Equivalence of Finite Automata”. Computer Science Technical Report TR71-114, Cornell University. 1971.

[Hopcroft & Ullman 1979] Hopcroft J., Ullman J. “Introduction to Automata Theory, Languages and Computation”. Mathematics and Computers in Simulation, 23(2):215, 1981.

[Hudson 1999] Hudson, S. "CUP Parser Generator for Java". www.cs.princeton.edu/~appel/modern/java/CUP, 1999.

[Ilie & Yu 2003] Ilie, L., Yu, S. "Follow Automata". Information and Computation 186. 140–162. 2003.

[Jalili 1983] Jalili, F. “A General Linear-Time Evaluator for Attribute Grammars”. ACM SIGPLAN Notices, 18(9): 35-44. 1983.

[Jelliffe 2001] Jelliffe, R. "The W3C XML Schema Specification in Context". O’Reilly XML.com. 2001.

[Jelliffe 2002] Jelliffe, R. "The Schematron Assertion Language 1.5". Academia Sinica Computing Centre. 2002.

[Johnson 1975] Johnson, S. C. “Yacc - Yet Another Compiler-compiler”. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ. 1975.

[Johnstone et al. 2004] Johnstone, A., Scott, E., Economopoulos, G. “The grammar Tool Box: A Case Study Comparing GLR Parsing Algorithms”. En Proceedings of the 4th Workshop on Language Descriptions, Tools and Applications. G. Hedin and E. V. Wick, eds. También en Electronic Notes in Theoretical Computer Science. Elsevier. 2004.

[Jourdan & Parigot 1991] Jourdan, M., Parigot, D. “Internals and Externals of the FNC-2 Attribute Grammar System”. Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science, Springer, 545: 485-504. 1991.

[Jourdan 1984] Jourdan, M. “Strongly Non-Circular Attribute Grammars and their Recursive Evaluation”. SIGPLAN symposium on Compiler construction, ACM SIGPLAN Notices, 19(6): 81-93. 1984.

[Kastens & Waite 1992] Kastens, U., Waite, W. "Modularity and Reusability in Attribute Grammars". Technical Report CUCS-613-92. University of Colorado. 1992.

[Kastens 1980] Kastens, U. “Ordered Attribute Grammars”. Acta Informatica 13: 229-256. 1980.

[Kastens et al. 1994] Kastens, U., Waite, W. M. “Modularity and Reusability in Attribute Grammars”. Acta Informatica, 31(7): 601-627. 1994.

[Kay 2007] Kay, M. "XSL Transformations (XSLT) Version 2.0". W3C Recommendation. 2007.

[Kennedy & Ramanathan 1979] Kennedy, K., Ramanathan, J. “A Deterministic Attribute Grammar Evaluator Based on Dynamic Sequencing”. ACM Transaction of Programming Languages and Systems 1(1): 142-160. 1979.

[Knuth 1965] Knuth D. E. “On the Translation of Languages from Left to Right”. Information and Control, 8(6): 607-639. 1965.

[Knuth 1968] Knuth, D. E. “Semantics of Context-free Languages”. Mathematical System Theory, 2(2): 127-145 / Math. System Theory, 5(1): 95-96. 1968.

[Koch & Scherzinger 2007] Koch, C. Scherzinger, S. “Attribute Grammars for Scalable Query Processing on XML Streams”. The VLDB Journal, 16(3): 317-342. 2007.

[Kodaganallur 2004] Kodaganallur, V. “Incorporating Language Processing into Java Applications: A JavaCC Tutorial”., IEEE Software, 21(4): 70-77. 2004.

[Lam et al. 2008] Lam, T.C., Ding, J.J., Liu, J.C. “XML Document Parsing: Operational and Performance Characteristics”. IEEE Computer 41(9): 30-37. 2008.

[Langendoen 1975] Langendoen, D. “Finite State Parsing of Phrase-structure Languages and the Status of Readjustment Rules in Grammar”. Linguistic Inquiry, 6(4):533-554. 1975.

[Lee et al. 2000] Lee, D., Chu, W.W. "Comparative Analysis of Six XML Schema Languages". ACM SIGMOD Record, 29(3), pp. 76-87. 2000.

[Leiss 1981] Leiss, E. "The Complexity of Restricted Regular Expressions and the Synthesis Problem for Finite Automata". Journal of Computer and System Sciences. Volume 23, Issue 3, pp:348–354. 1981.

[Leiss 1997] Leiss, E.L. "Solving Systems of Explicit Language Relations". Theoretical Computer Science 186(1-2): 83–105. 1997.

[Leiss 1999] Leiss, E. "Language Equations". Monographs in Computer Science. Springer. 1999.

[Lemke 1993] Lemke, I. Sander, G. “Visualization of Compiler Graphs”. Design report D 3.12.1-1, USAAR-1025-visual, ESPRIT Project #5399 Compare, Universität des Saarlandes, FB 14 Informatik, 1993.

[Magnusson & Hedin 2007] Magnusson, E., Hedin, G. “Circular Reference Attributed Grammars - Their Evaluation and Applications”. Science of Computer Programming, 68(1): 21-37. 2007.

[Mayr 2000] Mayr, R. "On the Complexity of Bisimulation Problems for Pushdown Automata". Proc. IFIP TCS’2000, Lecture Notes in Computer Science, Vol. 1872, Springer, Berlin, pp. 474–488. 2000.

[McLaughlin 2006] McLaughlin, B. "Java & XML". O’Reilly Media. 2006.

[Megginson et al. 1998] Megginson, D., Leventhal, M., Ducharme, R. "Structuring XML Documents". Prentice-Hall. 1998.

[Moreno-Ger et al. 2007] Moreno-Ger, P., Sierra, J.L., Martínez-Ortiz, I., Fernández-Manjón, B. "A Documental Approach to Adventure Game Development". Science of Computer Programming. 67(1): 3-31. 2007.

[Moreno-Ger et al. 2008] Moreno-Ger, P., Martínez-Ortiz, I., Sierra, J.L., Fernández-Manjón, B. "A Content-Centric Development Process Model". Computer. 41(3): 24-30. 2008.

[Murata 1995] Murata, M. "Forest Regular Languages and Tree Regular Languages". Technical Report. Fuji Xerox Systems. 1995.

[Murata 1999] Murata, M. "Hedge Automata: A Formal Model for XML Schemata". Technical Report. Fuji Xerox Systems. 1999.

[Murata 2000] Murata, M. "Regular Language Description for XML (RELAX)". ISO/IEC Technical Report DTR-22250-1. 2000.

[Murata et al. 2001] Murata, M., Lee, D., Mani, M. "Taxonomy of XML Schema Languages Using Formal Language Theory". Extreme Markup Languages. 2001.

[Murata et al. 2005] Murata, M., Lee, D., Mani, M., Kawaguchi, K. “Taxonomy of XML Schema Languages Using Formal Language Theory”. ACM Transactions on Internet Technology 5(4): 660-704. 2005.

[Nakano 2004] Nakano, K. "An Implementation Scheme for XML Transformation Languages Through Derivation of Stream Processors". Programming Languages and Systems: Second Asian Symposium (APLAS'04). Lecture Notes in Computer Science. Springer, 3302, pp. 74-90. 2004.

[Naur 1960] Naur, P. "Revised Report on the Algorithmic Language ALGOL 60". Communications of the ACM, 3(5): 299-314. 1960.

[Nederhof & Sarbo 1996] Nederhof, M. J., Sarbo, J. J. “Increasing the Applicability of LR Parsing. In Recent Advances in Parsing Technology”. H. Bunt and M. Tomita, eds. Kluwer Academic, Amsterdam, the Netherlands, 35-57. 1996.

[Nederhof 2000] Nederhof, M.J. “Regular Approximations of CFLs: A Grammatical View”. Advances in Probabilistic and other Parsing Text, Speech and Language Technology, 16: 221-241. 2000.

[Neven 1999] Neven, F. “Extensions of Attribute Grammars for Structured Document Queries”. En 7th International Workshop on Database Programming Languages, Lecture Notes In Computer Science, Springer, 99-116. 1999.

[Neven 2005] Neven, F. “Attribute Grammars for Unranked Trees as a Query Language for Structured Documents”. Journal of Computer and System Sciences, 70(2): 221-257. 2005.

[Nishimura & Nakano 2005] Nishimura, S., Nakano, K. “XML Stream Transformer Generation through Program Composition and Dependency Analysis”. Science of Computer Programming, 54(2-3): 257-290. 2005.

[Nozohoor-Farshi 1991] Nozohoor-Farshi, R. “GLR Parsing for e-grammars”. En Generalized LR Parsing, M. Tomita, ed. Kluwer Academic, Amsterdam, the Netherlands, 60-75. 1991.

[Okajima 2002] Okajima, D. “RelaxNGCC - Bridging the Gap Between Schemas and Programs”. vol.8, O’Reilly XML.com. 2002.

[Paakki 1995] Paakki, J. “Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation”. ACM Computer Surveys, 27(2): 196-255. 1995.

[Parr & Fisher 2011] Parr T., Fisher K. “LL(*): the Foundation of the ANTLR Parser Generator”. Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation: 425-436. 2011.

[Parr & Quong 1995] Parr T., Quong R. “ANTLR: A Predicated-LL(k) Parser Generator”. Software: Practice and Experience, 25(7): 789-810. 1995.

[Parr 2007] Parr, T. “The Definitive ANTLR Reference: Building Domain-Specific Languages”. Pragmatic Bookshelf. 2007.

[Paull & Unger 1968] Paull, M. C., Unger, S. H. "Structural Equivalence of Context-free Grammars". Journal of Computer and System Sciences, Volume 2, Issue 4, December, pp. 427-463. 1968.

[Psaila & Crespi-Reghizzi 1999] Psaila, G., Crespi-Reghizzi, S. “Adding Semantics to XML”. 2nd International Workshop on Attribute Grammars and their Applications, 113-132. 1999.

[Raggett 1999] Raggett, D. "Assertion Grammars". www.w3.org/People/Raggett/dtdgen/Docs. 1999.

[Rekers 1992] Rekers, J. G. “Parser Generation for Interactive Environments”. Ph.D. thesis, Universty of Amsterdam. 1992.

[Saraiva & Swiestra 1999] Saraiva, J., Swierstra, D. "Generic Attribute Grammars". Second Workshop of Attribute Grammars and their Applications, WAGA’99. Amsterdam, The Netherlands. 1999.

[Sarasa & Sierra 2013-a] Sarasa, A., Sierra, J.L. "Grammar-Driven Development of JSON Processing Applications". FedCSIS. 1545-1552. 2013.

[Sarasa & Sierra 2013-b] Sarasa, A., Sierra, J.L. "The Grammatical Approach: A Syntax-Directed Declarative Specification Method for XML Processing Tasks". Computer Standards & Interfaces 35(1): 114-131. 2013.

[Sarasa & Sierra 2015] Sarasa, A., Sierra, J.L. "A Syntax-Directed Model Transformation Framework based on Attribute Grammars". SLATE'15. 2015.

[Sarasa 2012] Sarasa, A. “Desarrollo de Aplicaciones XML Mediante Herramientas de Construcción de Procesadores de Lenguaje”. Tesis Doctoral. e-prints UCM. 2012.

[Sarasa et al. 2008] Sarasa, A., Navarro, I., Sierra, J.L, Fernández-Valmayor, A. “Building a Syntax Directed Processing Environment for XML Documents by Combining SAX and JavaCC”. 3rd Int. Workshop on XML Data Management Tools & Techniques. DEXA’08. September 1-5, Turin, Italy. 2008.

[Sarasa et al. 2009-a] Sarasa, A., Temprado, B., Martínez, A., Sierra, J.L., Fernández-Valmayor, A. “Building an Enhanced Syntax-Directed Processing Environment for XML Documents by Combining StAX and CUP”. Fourth Int. Workshop on Flexible Database and Information Systems. DEXA’09. August 31 – September 4, Linz, Austria. 2009.

[Sarasa et al. 2009-b] Sarasa, A., Temprado, B., Sierra, J.L. Fernández-Valmayor, A. “XML Language-Oriented Processing with XLOP”. Proceedings of the 5th International Symposium on Web and Mobile Information Services (AINA’09 Workshops). IEEE Computer Society. 2009.

[Sarasa et al. 2011] Sarasa, A., Temprado, B., Sierra, J.L. "Engineering Web Services with Attribute Grammars: A Case Study". ACM SIGSOFT Software Engineering Notes 36(1): 1-8. 2011.

[Sarasa et al. 2012] Sarasa A., Temprado B., Rodriguez D., Sierra J.L., "Building XML-driven Application Generators with Compiler Construction Tools". Comput. Sci. Inf. Syst. 9(2): 485-504. 2012.

[Schreiner & Friedman 1985) Schreiner, A.T., Friedman, H.G. "Introduction to Compiler Construction with UNIX". Prentice-Hall. 1985.

[Shaban 1994] Shaban, M. "A Hybrid GLR Algorithm for Parsing with Epsilon Grammars". Boston University, Technical Report. 1994.

[Sierra et al. 2006] Sierra, J. L. Fernández-Valmayor, A. Guinea, M. Hernánz, H. "From Research Resources to Virtual Objects: Process model and Virtualization Experiences". Journal of Educational Technology & Society, 9(3), pp. 56-68. 2006

[Sierra et al. 2008] Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B. "From Documents to Applications Using Markup Languages". IEEE Software 25(2), 68-76. 2008.

[Stanchfield 2005] Stanchfield, S. "ANTXR: Easy XML Parsing based on The ANLR Parser Generator". Java Due.com, Hillcrest Comm. & FGM, Inc. javadude.com/tools/antxr/index.html. 2005.

[Takahashi 1975] Takahashi, M. "Generalizations of Regular Sets and Their Application to a Study of Context-Free Languages". Information and Control, 27(1), pp. 1-36. 1975.

[Tarjan 1972] Tarjan, R. E. "Depth-first Search and Linear Graph Algorithms". SIAM Journal on Computing 1(2): 146-160. 1972.

[Temprado 2009] Temprado, B. "XLOP (XML Language-Oriented Processing)". Proyecto Fin de Carrera. e-prints UCM. 2009.

[Temprado 2010] Temprado, B. "Desarrollo Dirigido por Lenguajes de Aplicaciones de Procesamiento XML: Especificación Modular Basada en Gramáticas de Atributos y Generación Automática". Proyecto Fin de Máster. e-prints UCM. 2010.

[Temprado et al. 2010-a] Temprado, B., Sarasa, A., Sierra, J.L. "Modular Specifications of XML Processing Tasks with Attribute Grammars defined on Multiple Syntactic Views". Fifth International Workshop on Flexible Database and Information Systems Technology. FlexDBIST. 2010.

[Temprado et al. 2010-b] Temprado, B., Sarasa, A., Sierra, J.L. "Managing the Production and Evolution of e-Learning Tools with Attribute Grammars". The 10th IEEE International Conference on Advanced Learning Technologies. ICALT. 2010.

[Temprado et al. 2011] Temprado, B., Sarasa, A., Sierra, J.L. "Checking the Conformance of Grammar Refinements with Respect to Initial Context-Free Grammars". Proc. Of FedCSIS 2011. IEEE Computer Society, 887-890. 2011.

[Thatcher 1967] Thatcher, J.W. "Characterizing Derivation Trees of Context-Free Grammars Through a Generalization of finite automata theory". Journal of Computer and Systems Science, 1(4), pp. 317-322. 1967.

[Thatcher 1973] Thatcher, J.W. "Tree Automata: An Informal Survey". Currents in Theory of Computing. Prentice-Hall, pp. 143–172. 1973.

[Thompson 1968] Thompson, K. “Regular Expression Search Algorithm”. Communications of the ACM, 11(6): 365–375. 1968.

[Thompson et al. 2001] Thompson, H.S, Beech, D., Maloney, M., Mendelsohn, N. "XML Schema Part 1: Structures". W3C Recommendation. 2001.

[Tomita 1985] Tomita, M. “Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems”. The Springer International Series in Engineering and Computer Science. 1985.

[Tomita 1986] Tomita, M. “Efficient Parsing for Natural Language”. Kluwer Academic, Boston. 1986.

[Tozawa 2001] Tozawa, A. "Towards Static Type Checking for XSLT". ACM Symposium on Document Engineering(DocEng’01). ACM, pp. 18-27. 2001.

[Visser 1997] Visser, E. “Syntax Definition for Language Prototyping”. Ph.D. thesis, University of Amsterdam. 1997.

[Vlist 2001] Vlist, E. "Comparing XML Schema Languages". O’Reilly XML.com. 2001.

[Vogt et al. 1989] Vogt, H., Swierstra, S., Kuiper, M. "Higher-Order Attribute Grammars". Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation. 1989.

[Wilhelm 1982] Wilhelm, R. “LL- and LR-Attributed Grammars”. Fachtagung über Programmiersprachen, Informatik-Fachberichte, Springer, 53: 151-164. 1982.

[Wyk et al. 2010] Wyk, E.V., Bodin, D., Gao, J., Krishnan, L. “Silver: An Extensible Attribute Grammar System”. Science of Computer Programming, 75(1-2): 39-54. 2010.

[yComp] Kroll, M., Braun, M., Beck, M., Geiss, R., Hack, S., Leiss, P. “VCG Viewer and Compiler Graph Visualization Tool”. pp.ipd.kit.edu/firm/yComp.

[Younger 1967] Younger, D. H. “Recognition and Parsing of Context-free Languages in time n3”. Information and Control, 10(2): 189-208. 1967.